Χρήση του μηχανισμού των futexes για υποστήριξη εφαρμογών OpenMP

Διονύσιος Χρονόπουλος

Διπλωματική Εργασία

Επιβλέπων: Βασίλειος Δημακόπουλος

Πανεπιστήμιο Ιωαννίνων

Τμήμα Μηχ. Η/Υ και Πληροφορικής

Οκτώβριος 2025

Περιεχομενα

Περιληψη

Χρήση του μηχανισμού των futexes για υποστήριξη εφαρμογών OpenMP.
Επιβλέπων: Βασίλειος Δημακόπουλος.

Η παρούσα Διπλωματική Εργασία αφορά την μελέτη της ιδέας και του τρόπου χρήσης του μηχανισμού των `futexes` (Fast Userspace Mutex). Ο μηχανισμός αυτός αποτελεί θεμέλιο για την επίτευξη του συγχρονισμού μεταξύ νημάτων και προτιμάται λόγω της ευελιξίας που προσφέρει. Με την ένταξή του στον πυρήνα του Linux ξεπεράσθηκαν πολλά από τα προβλήματα των προϋπάρχοντων λειτουργιών συγχρονισμού νημάτων και δόθηκε μεγαλύτερη προγραμματιστική ελευθερία - αλλά και ευθύνη - στον προγραμματιστή. Η σωστή χρήση τους αποσκοπεί στην επίτευξη υψηλών επιδόσεων σε πολυνηματικές εφαρμογές που υλοποιούνται σε πολυπύρηνα συστήματα, τα οποία σήμερα είναι αναμφίβολα ευρέως διαδεδομένα.

Επιπλέον, στα πλαίσια της εργασίας αυτής ερευνάται και η ενσωμάτωση του εν λόγω μηχανισμού στη λειτουργία ενός μεγάλης κλίμακας προγραμματιστικού εργαλείου ανοιχτού κώδικα, τον παραλληλοποιητικό μεταφραστή `OMPi`. Με την ενσωμάτωσή του, ο μεταφραστής είναι σε θέση να αντικαθιστά τις διάφορες μεθόδους συγχρονισμού νημάτων με ισοδύναμες που χρησιμοποιούν futexes. Οι μέθοδοι συγχρονισμού νημάτων που χρησιμοποιούν futexes, οι τεχνικές λεπτομέρειες της διαδικασίας ενσωμάτωσης των μεθόδων αυτών στο περιβάλλον του OMPi και η εκτίμηση των επιδόσεων των μεθόδων συγχρονισμού αποτελούν αντικείμενα της εργασίας.

Abstract in English

Use of futexes for the runtime support of OpenMP programs.
Advisor: Vassilios Dimakopoulos.

This Diploma Thesis is about the study of the idea and the way of using the mechanism of `futexes` (Fast Userspace Mutex). This mechanism is a foundation for achieving synchronization between threads and is preferred because of the flexibility it offers. Its inclusion in the Linux kernel overcame many of the problems of pre-existing thread synchronization features and gave greater programming freedom - and responsibility - to the programmer. Their proper use is aimed at achieving high performance in multi-threaded applications implemented on multi-core systems, which are undoubtedly widespread today.

Moreover, in the context of this work, the integration of this mechanism into the operation of a large-scale open source programming tool, the parallelizing `OMPi` compiler, is also investigated. With its integration, the compiler is able to replace the various thread synchronization methods with equivalent ones that use futexes. This thesis examines thread synchronization methods based on futexes, their integration into the OMPi environment and the performance evaluation of these synchronization techniques.

Κεφαλαιο 1

Εισαγωγη

Το πρώτο κεφάλαιο αποτελεί μία παρουσίαση του τομέα της παράλληλης επεξεργασίας, τον τομέα που ανήκει η παρούσα διπλωματική εργασία, που μπορεί να αναλυθεί στα παράλληλα συστήματα και τον παράλληλο προγραμματισμό. Ύστερα, ακολουθεί μια λεπτομερής περιγραφή του αντικειμένου της εργασίας και κατόπιν μια επισκόπηση της δομής της παρούσας εργασίας και από τα κεφάλαια από τα οποία αποτελείται.

1.1 Σύντομη παρουσίαση των παράλληλων συστημάτων

Οι επεξεργαστές ενός ηλεκτρονικού υπολογιστή ανήκουν στην κατηγορία των μικροεπεξεργαστών, των ολοκληρωμένων κυκλωμάτων που εκτελούν λογική ελέγχου αλλά και επεξεργασία δεδομένων. Στη σημερινή εποχή, οι μικροεπεξεργαστές έχουν εξελιχθεί όχι μόνο σε γρήγορους επεξεργαστές οικιακής ή επιστημονικής χρήσης, αλλά πλέον αποτελούν πολυεπεξεργαστικά συστήματα — κάθε τσιπ (γνωστό

και ως ”υποδοχή” ή *socket*) περιλαμβάνει πολλούς επεξεργαστές ή πυρήνες (*cores*). Κάθε πυρήνας διαθέτει πολλαπλά επίπεδα κρυφής μνήμης και αρκετούς ”λογικούς” επεξεργαστές (*logical processors*), οι οποίοι μοιράζονται τις μονάδες εκτέλεσης. Από το 2010 και μετά, είναι σύνηθες ακόμη και ένας φορητός υπολογιστής να ενσωματώνει 2 ή 4 πυρήνες, με κάθε πυρήνα να υποστηρίζει 2 νήματα υλικού, προσφέροντας συνολικά 4 έως 8 λογικούς επεξεργαστές.

Παρότι οι επεξεργαστές προσφέρουν πολύ καλές επιδόσεις για τις περισσότερες υπολογιστικές εργασίες, υπάρχει κι αυξανόμενη ανάγκη για εκτεταμένους αριθμητικούς υπολογισμούς. Άλλο παράδειγμα μικροεπεξεργαστών αποτελούν οι Μονάδες Επεξεργασίας Γραφικών (GPUs), που προσφέρουν αποδοτική επεξεργασία μεγάλων πινάκων δεδομένων μέσω τεχνικών SIMD (Single Instruction Multiple Data), που πρωτοεμφανίστηκαν στους ακριβούς και μεγάλους σε έκταση υπερυπολογιστές. Παράλληλα, οι ίδιοι οι Κεντρικοί Επεξεργαστές ή Κεντρικές Μονάδες Επεξεργασίας (CPUs) ενσωματώνουν όλο και πιο ισχυρές τεχνολογίες στις συνήθεις αρχιτεκτονικές x86 και AMD64, δίνοντάς τους τη δυνατότητα να εκτελούν αποτελεσματικά εργασίες σε μεγάλους πίνακες δεδομένων ή εκτενή υπολογιστικά φύλλα.

Παρόλο που οι σημερινές τεχνολογίες και επιδόσεις των επεξεργαστών είναι εντυπωσιακές, χρειάστηκε πολύς χρόνος για να επιτευχθούν. Υπήρξε μάλιστα μια περίοδος, γύρω στο 2005, κατά την οποία φαινόταν αδύνατη οποιαδήποτε ουσιαστική βελτίωση στην απόδοση των μικροεπεξεργαστών. Τότε εμφανίστηκε το λεγόμενο ”power wall” — μια κατάσταση όπου η αυξημένη πυκνότητα τρανζίστορ σε ένα τσιπ εμπόδιζε την περαιτέρω αύξηση της συχνότητας ρολογιού, καθώς αυτή συνεπαγόταν υπερβολική κατανάλωση ισχύος και σημαντική επιδείνωση των θερμικών χαρακτηριστικών του επεξεργαστή.

Το πρόβλημα αυτό ξεπεράστηκε χάρη στη στροφή προς την παράλληλη επεξεργασία. Σήμερα, σχεδόν όλοι οι υπολογιστές — είτε πρόκειται για υπερυπολογιστές, εξυπηρετητές, είτε για έξυπνα κινητά τηλέφωνα — βασίζονται σε πολυπύρηνους επεξεργαστές. Αυτοί προσφέρουν υψηλές επιδόσεις ανά μονάδα ισχύος, με διαχειρίσιμο κόστος. Η υλοποίηση της παράλληλης επεξεργασίας σε ένα υπολογιστικό σύστημα μπορεί να διαφέρει, ωστόσο οι βασικές αρχιτεκτονικές της προσεγγίσεις συνοψίζονται παρακάτω.

1.1.1 Συστήματα κοινόχρηστης μνήμης (Shared memory)

Τα συστήματα με κοινόχρηστη μνήμη αποτελούνται από πολλούς επεξεργαστές που έχουν πρόσβαση σε μία ενιαία μνήμη μέσω ενός δικτύου διασύνδεσης που εξασφαλίζει τον συγχρονισμό μεταξύ τους. Η μνήμη μπορεί να είναι κατανεμημένη σε επιμέρους μονάδες (modules), αλλά όλοι οι επεξεργαστές βλέπουν μόνο έναν ενιαίο, κοινόχρηστο χώρο διευθύνσεων. Η επικοινωνία και η συνεργασία μεταξύ επεξεργαστών επιτυγχάνεται μέσω της ανάγνωσης και εγγραφής κοινών μεταβλητών στη μνήμη.

Το δίκτυο που συνδέει επεξεργαστές και μνήμη μπορεί να είναι ένας απλός δίαυλος, ένα πιο σύνθετο διακοπτικό δίκτυο (όπως crossbar) ή ακόμα και ένα πολυεπίπεδο δίκτυο, όπως το δίκτυο Δέλτα. Όταν η διασύνδεση βασίζεται σε δίαυλο και δεν υπάρχει κάποια αυστηρή ιεραρχία μεταξύ των επεξεργαστών, τότε το σύστημα ονομάζεται συμμετρικός πολυεπεξεργαστής (Symmetrical MultiProcessor ή SMP). Σε αυτά τα συστήματα, όλοι οι επεξεργαστές έχουν ισότιμη πρόσβαση στην κοινόχρηστη μνήμη, χαρακτηριστικό που τα κατατάσσει στα συστήματα ομοιόμορφης προσπέλασης μνήμης (Uniform Memory Access ή UMA).

Στην πράξη, οι σύγχρονοι πολυπύρηνοι επεξεργαστές ακολουθούν αυτή τη φιλοσοφία, ενσωματώνοντας πολλαπλούς πυρήνες και κρυφές μνήμες (caches) πολλών επιπέδων, ώστε να μειώνεται η καθυστέρηση πρόσβασης στην κύρια μνήμη. Ωστόσο, η απόδοση του συστήματος περιορίζεται από το εύρος ζώνης του διαύλου· όσο αυξάνεται ο αριθμός των επεξεργαστών, τόσο εντείνεται ο ανταγωνισμός για πρόσβαση στη μνήμη, οδηγώντας σε συμφόρηση και καθυστερήσεις. Παράλληλα, αυξάνεται και η πολυπλοκότητα των μηχανισμών συγχρονισμού και διαχείρισης μνήμης. Γι' αυτόν τον λόγο, για να επιτύχουμε καλύτερη κλιμάκωση, γίνεται χρήση κρυφών μνημών ή πιο εξελιγμένων δικτύων διασύνδεσης.

1.1.2 Συστήματα κατανεμημένης μνήμης (Distributed memory)

Στα συστήματα κατανεμημένης μνήμης, η υπολογιστική ισχύς διανέμεται σε πολλές επεξεργαστικές μονάδες, που ονομάζονται κόμβοι. Κάθε κόμβος περιλαμβάνει έναν ή περισσότερους επεξεργαστές καθώς και τη δική του τοπική μνήμη, στην οποία έχει απευθείας πρόσβαση μόνο ο ίδιος. Η επικοινωνία μεταξύ των κόμβων γίνεται μέσω κάποιου δικτύου διασύνδεσης, όπως το Ethernet.

Αν και κάθε κόμβος έχει τη δική του μνήμη, είναι δυνατή η απομακρυσμένη

πρόσβαση στη μνήμη άλλων κόμβων μέσω ανταλλαγής μηνυμάτων. Σε πιο μεγάλης κλίμακας παράλληλα συστήματα, είναι δυνατό οι ίδιοι οι κόμβοι να είναι πολυεπεξεργαστικά συστήματα τύπου SMP. Σε αυτές τις περιπτώσεις, μπορούμε να δημιουργήσουμε έναν ενιαίο χώρο διευθύνσεων, υπό την προϋπόθεση ότι εφαρμόζονται κατάλληλα πρωτόκολλα συνοχής μνήμης ώστε να διατηρείται η εγκυρότητα των δεδομένων.

Αυτού του τύπου τα συστήματα αναφέρονται ως «κατανεμημένη κοινόχρηστη μνήμη» (Distributed Shared Memory ή DSM), ενώ λόγω της μη ενιαίας πρόσβασης σε όλα τα τμήματα της μνήμης με τον ίδιο χρόνο, χαρακτηρίζονται και ως συστήματα μη-ομοιόμορφης προσπέλασης μνήμης (Non-Uniform Memory Access – NUMA). Όταν υπάρχουν ιεραρχικές και πολυεπίπεδες κρυφές μνήμες στους κόμβους, απαιτείται η χρήση ειδικού πρωτοκόλλου συνοχής για τις caches (πρωτόκολλα συνοχής κρυφής μνήμης ή cache coherence protocols), ώστε να εξασφαλίζεται ότι κάθε πρόσβαση στη μνήμη επιστρέφει την πιο πρόσφατη τιμή. Τέτοια συστήματα καλούνται "cache-coherent NUMA" ή ccNUMA. Σήμερα, τα περισσότερα συστήματα τύπου NUMA είναι και ccNUMA.

Ένα σύγχρονο παράδειγμα συστήματος κατανεμημένης μνήμης είναι οι υπολογιστικές συστάδες (compute clusters), που αποτελούνται από ομάδες υπολογιστών εξοπλισμένων με εξειδικευμένους επεξεργαστές πολλών πυρήνων, GPUs γενικής χρήσης (GPGPUs), ή άλλους επιταχυντές. Οι υπολογιστές αυτοί διασυνδέονται με δίκτυα πολύ υψηλού εύρους ζώνης και εξαιρετικά χαμηλής καθυστέρησης.

1.2 Σύντομη παρουσίαση του παράλληλου προγραμματισμού

Την ώρα που η τεχνολογία των μικροεπεξεργαστών εξελίσσοταν και υιοθετούνταν η ιδέα της παράλληλης επεξεργασίας, οι προγραμματιστές παρέμεναν ενήμεροι για τις εν λόγω αλλαγές και φρόντιζαν να τις λαμβάνουν υπόψιν τους στα έργα τους.

Αν και είναι δυσεύρετο πλέον ένα μονοπύρηνο σύστημα, τα *μοντέλα προγραμματισμού* δεν έχουν εξελιχθεί ώστε η πλήρης αξιοποίηση όλων των πυρήνων και υπολογιστικών πόρων του συστήματος να γίνεται αυτόματα. Ως μοντέλο παράλληλου προγραμματισμού εννοούμε τον τρόπο με τον οποίο ο προγραμματιστής αντιλαμβάνεται τη λειτουργία του υπολογιστή. Κατ' επέκταση, ο παράλληλος προγραμματισμός παραμένει ένα μη-τετριμμένο πρόβλημα και οι λύσεις που υιοθετούνται

δεν είναι καθολικές και διαφέρουν ανάλογα με την αρχιτεκτονική του συστήματος όπου εκτελείται ο κώδικας.

Για την ακρίβεια, στο σειριακό μοντέλο ο προγραμματιστής δεν γνώριζε τις αρχιτεκτονικές λεπτομέρειες της μνήμης και του επεξεργαστή. Όμως, το εντελώς αντίθετο συμβαίνει με τον προγραμματισμό των παράλληλων μηχανών και συστημάτων. Στη γενική περίπτωση, ο μοντέρνος προγραμματιστής που επιθυμεί να παραλληλοποιήσει τις εργασίες που εκτελεί μια εφαρμογή ή γενικά να εκμεταλλευτεί καλύτερα τις παρεχόμενες δυνατότητες, κάνει μια θεώρηση των διαθέσιμων υπολογιστικών πόρων και υιοθετεί ένα υπάρχον μοντέλο παράλληλου προγραμματισμού, σύμφωνα με την περίσταση. Ακολουθεί μια παρουσίαση των σημαντικότερων μοντέλων παράλληλου προγραμματισμού.

1.2.1 Μοντέλα κοινόχρηστης μνήμης (Shared memory)

Ο προγραμματισμός σε περιβάλλοντα κοινόχρηστης μνήμης βασίζεται στη χρήση νημάτων (threads) ή ινών (fibers). Τα νήματα είναι ανεξάρτητες μονάδες εκτέλεσης, με τη δική τους στοίβα και μετρητή προγράμματος, που μοιράζονται έναν κοινό χώρο διευθύνσεων. Από την άλλη, οι ίνες αποτελούν μια ειδική μορφή νημάτων που υλοποιούν συνεργατική πολυεκτέλεση. Σε αυτή την περίπτωση, το λειτουργικό σύστημα δεν διακόπτει την εκτέλεσή τους αυτόματα, γεγονός που αποφεύγει τα δαπανηρά context switches – δηλαδή τις λειτουργίες αλλαγής μεταξύ διαφορετικών νημάτων. Αντί για αυτό, το "κατάστασή" τους αποθηκεύεται και μπορεί να επανενεργοποιηθεί αργότερα χωρίς επανεκκίνηση της στοίβας ή του μετρητή προγράμματος.

Η χρήση πολυνηματικού προγραμματισμού εισάγει κινδύνους σχετικά με την εγκυρότητα των κοινόχρηστων δεδομένων. Συγκεκριμένα, δημιουργούνται καταστάσεις γνωστές ως «συνθήκες ανταγωνισμού» (race conditions), όπου δύο ή περισσότερα νήματα προσπαθούν να προσπελάσουν ή να τροποποιήσουν τα ίδια δεδομένα στη μνήμη ταυτόχρονα. Αυτές οι καταστάσεις οδηγούν σε μη-ντετερμινιστική συμπεριφορά του προγράμματος, καθιστώντας τα σφάλματα δύσκολα στον εντοπισμό και στην αναπαραγωγή.

Για την αποφυγή τέτοιων προβλημάτων, χρησιμοποιούνται τεχνικές όπως η ατομικότητα (atomicity) και ο αμοιβαίος αποκλεισμός (Mutual Exclusion ή mutex). Η πρώτη βασίζεται στη χρήση ειδικών εντολών που εκτελούνται αδιαίρετα σε επίπεδο

υλικού, ενώ η δεύτερη στηρίζεται σε μηχανισμούς, όπως τα mutexes, που εξασφαλίζουν αποκλειστική πρόσβαση στα κοινόχρηστα δεδομένα κατά την εγγραφή.

Η πιο διαδεδομένη βιβλιοθήκη για πολυνηματικό προγραμματισμό είναι η POSIX (Portable Operating System Interface) threads (pthreads ή Pthreads), που είναι διαθέσιμη σε όλα τα λειτουργικά συστήματα συμβατά με το πρότυπο POSIX. Η βιβλιοθήκη αυτή προσφέρει όλα τα απαραίτητα εργαλεία για τη δημιουργία και διαχείριση νημάτων – από την έναρξη και τερματισμό έως συγχρονισμό μέσω mutexes, μεταβλητών συνθήκης (condition variables) και άλλων μηχανισμών. Ωστόσο, η διαχείριση όλων αυτών των διεργασιών – δημιουργία, συντονισμός, συγχρονισμός και συλλογή αποτελεσμάτων – επαφίεται αποκλειστικά στον προγραμματιστή. Αυτό παρέχει αυξημένο έλεγχο στη ροή και την απόδοση του προγράμματος, αλλά ταυτόχρονα αυξάνει τον κίνδυνο εμφάνισης σφαλμάτων.

Σαν απάντηση στο πρόβλημα αυτό, το OpenMP (Open Multi-Processing) είναι μια βιβλιοθήκη-διεπαφή προγραμματιστικών εφαρμογών (Application Programming Interface, API) πολυνηματισμού υψηλού επιπέδου για συστήματα κοινόχρηστης μνήμης, που διευκολύνει την ανάπτυξη παράλληλου κώδικα μέσω οδηγιών (directives) προς τον μεταφραστή, ρουτινών και μεταβλητών περιβάλλοντος. Ενσωματώνεται εύκολα σε υπάρχοντα σειριακά προγράμματα, χωρίς να απαιτούνται σημαντικές αλλαγές στον αρχικό κώδικα. Υποστηρίζεται από μεταφραστές C/C++ μέσω των οδηγιών που αρχίζουν με το πρόθεμα #pragma στον πηγαίο κώδικα, οι οποίες οδηγίες αγνοούνται από όσους μεταφραστές δεν υποστηρίζουν την βιβλιοθήκη. Τέλος, το runtime σύστημα του OpenMP διαχειρίζεται την εκτέλεση και το συντονισμό των νημάτων, μειώνοντας την πολυπλοκότητα και καθιστώντας τον πολυνηματισμό προσβάσιμο ακόμη και σε μη έμπειρους προγραμματιστές, διευκολύνοντας την επίτευξη παράλληλης επιτάχυνσης εφαρμογών.

1.2.2 Μοντέλα κατανεμημένης μνήμης (Distributed memory)

Ο προγραμματισμός σε συστήματα κατανεμημένης μνήμης βασίζεται στην ανταλλαγή μηνυμάτων μεταξύ ανεξάρτητων διεργασιών που εκτελούνται σε διαφορετικούς κόμβους. Λόγω της φύσης της αρχιτεκτονικής, δεν υπάρχουν κοινόχρηστες μεταβλητές· οι κόμβοι επικοινωνούν μεταξύ τους για να ενημερώσουν σχετικά με αλλαγές σε κοινά δεδομένα.

Ο συγχρονισμός είναι απαιτητικός, γι' αυτό και συχνά διαχωρίζονται οι υπολογι-

στικές από τις επικοινωνιακές λειτουργίες. Ο προγραμματιστής πρέπει να χειριστεί σωστά τη ροή των μηνυμάτων, γνωρίζοντας αν η επικοινωνία είναι σύγχρονη (τύπου ''ραντεβού'', δηλαδή αν η επικοινωνία μπλοκάρει μέχρι να ληφθεί το μήνυμα) ή ασύγχρονη (η επικοινωνία συνεχίζεται ανεξαρτήτως παραλαβής). Μπορεί να δίνεται και η δυνατότητα για συλλογική επικοινωνία, η οποία αναφέρεται σε επικοινωνίες μεταξύ πολλών κόμβους ή διεργασιών ταυτόχρονα, για την αποτελεσματική συνεργασία μεταξύ των διεργασιών. Κάποια είδη συλλογικής επικοινωνίας αποτελούν η Διασκόρπιση (Scatter) για την αποστολή διαφορετικών μηνυμάτων από μία διεργασία προς όλες τις άλλες, καθώς και Συλλογή (Gather)όταν μία διεργασία λαμβάνει ένα μήνυμα από κάθε μία από τις υπόλοιπες.

Αναμφίβολα, η επικοινωνία είναι δαπανηρή, και συνιστάται ελαχιστοποίηση των ανταλλαγών. Το πρότυπο MPI (Message Passing Interface) είναι το πιο διαδεδομένο εργαλείο για τέτοιου τύπου επικοινωνία, με υλοποιήσεις όπως OpenMPI, MPICH και Intel MPI, προσφέροντας υψηλού επιπέδου δυνατότητες στις γλώσσες C/C++ και Fortran, χωρίς να απαιτείται σε βάθος γνώση του δικτυακού προγραμματισμού.

1.3 Αντικείμενο της Διπλωματικής εργασίας

Εστιάζοντας στο γεγονός πως σε ένα μοντέλο κοινόχρηστης μνήμης συχνά απαιτείται συγχρονισμός μεταξύ των νημάτων, αντικείμενο της παρούσας εργασίας αποτελεί η μελέτη της ιδέας και της χρήσης του μηχανισμού των *futexes* (Fast UserSpace Mutex, ενικός: *futex*), με σκοπό την βελτίωση απόδοσης των μηχανισμών συγχρονισμού. Ο όρος ''futex'' αποτελεί ταυτόχρονα μια κλήση συστήματος (`futex call`) αλλά και μια μεταβλητή (`futex-word`) η οποία χρησιμοποιείται συνδυαστικά με την κλήση για τη δημιουργία δομών και μεθόδων συγχρονισμού νημάτων, αλλά και διεργασιών. Μια πρώιμη μορφή του μηχανισμού παρουσιάστηκε το 2002 [1] κι εντάχθηκε για πρώτη φορά στον πυρήνα του Linux (Linux Kernel) στην έκδοση 2.5.4 [2]. Η λειτουργικότητά του έχει επεκταθεί από τότε και η έκδοση που χρησιμοποιείται πλέον είναι εκείνη που εντάχθηκε στον πυρήνα στην έκδοση 2.6.7.

Θεωρείται γνωστό πως η λύση του προβλήματος των συνθηκών ανταγωνισμού αποτελεί ο αμοιβαίος αποκλεισμός ή η χρήση `mutexes` ισοδύναμα. Πολύ συχνά χρησιμοποιούνται σε συνδυασμό με μεταβλητές συνθήκης (`condition variables`). Ο τρόπος υλοποίησής τους είναι κρίσιμης σημασίας διότι εκτός από λόγους ορθότη-

τας, υπάρχουν λόγοι απόδοσης και ρυθμού διεκπεραίωσης. Το σύνηθες σύγχρονο σενάριο χρήσης τους είναι το εξής: ένα νήμα Α αποκτά (ή κλειδώνει) τη κλειδαριά κι έτσι αποκτά αποκλειστική πρόσβαση στον κοινόχρηστο πόρο. Ύστερα, αν δεν ισχύει κάποια συνθήκη, τότε ξεκλειδώνει την κλειδαριά και κοιμίζεται. Στο μεταξύ, ένα άλλο νήμα Β μπορεί να αποκτήσει πρόσβαση στον κοινόχρηστο πόρο και πράττει όπως το Α, ενώ παράλληλα το νήμα Α αναμένει το σήμα αφύπνισης. Βέβαια, ο τρόπος υλοποίησης του σεναρίου με κώδικα διαφέρει από υλοποίηση σε υλοποίηση.

Για παράδειγμα, η προηγούμενη υλοποίηση της pthreads στο Linux αποτελούσε η *LinuxThreads*. Το μεγαλύτερο πρόβλημα της υλοποίησης αυτής ήταν η έλλειψη αποτελεσματικών μηχανισμών συγχρονισμού, που ανάγκαζε την υλοποίηση να χρησιμοποιεί σήματα. Η χρήση σημάτων για την υλοποίηση των μηχανισμών συγχρονισμού προκαλεί σοβαρά προβλήματα. Η καθυστέρηση των λειτουργιών είναι μεγάλη και συχνά εμφανίζονται *ψευδείς αφυπνίσεις* (*spurious wakeups*), οι οποίες πρέπει να διαχωρίζονται από τις κανονικές και να αντιμετωπίζονται κατάλληλα [3]. Εκτός από τον κίνδυνο εσφαλμένης ερμηνείας αφύπνισης, ο ίδιος ο διαχωρισμός επιβαρύνει επιπλέον το σύστημα σημάτων του πυρήνα. Για αυτόν τουλάχιστον το λόγο, η τωρινή υλοποίηση του προτύπου POSIX thread library έχει αντικατασταθεί κι ονομάζεται *NPTL* (*Native POSIX Thread Library*), στα πλαίσια της οποίας υπάρχει και λειτουργεί ο μηχανισμός των futexes.

Άλλο παράδειγμα - κίνητρο δημιουργίας των futexes - αποτελεί το σύστημα διαδεργασιακής επικοινωνίας του SysV (*System V Inter Process Communication ή IPC*) των παραδοσιακών συστημάτων UNIX. Αυτό το σύστημα βασίζεται σε μηχανισμούς όπως οι σημαφόροι, ουρές μηνυμάτων και sockets για την επίτευξη διαδιεργασιακής επικοινωνίας. Βέβαια, οι εν λόγω μηχανισμοί πρέπει να υλοποιούνται μέσω κλήσεων συστήματος (system calls, π.χ. semop()). Το μειονέκτημα αυτής της προσέγγισης είναι ότι κάθε πρόσβαση στο κλείδωμα απαιτεί system call κι όταν η συμφόρηση (contention) στα κλειδώματα είναι μικρή, το υπολογιστικό κόστος χρήσης ενός system call μπορεί να είναι σημαντικό.

Όπως θα εξηγηθεί σε επόμενο κεφάλαιο, οι κλήσεις συστήματος δεν είναι απαραίτητες κάθε φορά που ένα νήμα αιτείται πρόσβαση σε μια κρίσιμη περιοχή. Για αυτό, ο μηχανισμός των futexes προσφέρει έναν εύχρηστο τρόπο επικοινωνίας μεταξύ προγραμματιστή και πυρήνα λειτουργικού συστήματος, με σκοπό την ασφαλή κοίμηση κι αφύπνιση νημάτων μόνο όποτε απαιτείται από τον μηχανισμό συγχρονισμού νημάτων (πχ. mutex, cond. variable κ.ά.) που κατασκευάζει ο προγραμμα-

τιστής.

Δυστυχώς, αν θέλουμε να αποφύγουμε ολοκληρωτικά τις κλήσεις συστήματος, η άλλη επιλογή είναι πάντα η αναποτελεσματική μέθοδος του *busy-waiting ή busy-loop* (δηλαδή συνεχή έλεγχο του mutex και σπατάλη κύκλων της CPU, βλ. Αλγόριθμοι Dekker ή Peterson). Εντούτοις, αν και η επέμβαση του λειτουργικού συστήματος είναι συχνά εξίσου κοστοβόρα όσο ένα context-switch, τα futexes δημιουργήθηκαν για να συμβάλλουν στη μείωση αυτών των επεμβάσεων στην διαδικασία συγχρονισμού νημάτων.

Η χρησιμότητά του εν λόγω μηχανισμού εκτιμήθηκε πολύ, καθώς σε αυτόν τον μηχανισμό βασίζονται πολλοί άλλοι. Στο Linux, πολλές από τις κλήσεις τις οποίες παρέχει η pthreads χρησιμοποιούν τον μηχανισμό των futexes. Επιπλέον, `futexes` χρησιμοποιούνται από την βιβλιοθήκη χρόνου εκτέλεσης της OpenMP που ονομάζεται *libgomp* (*GNU Offloading and Multi Processing Runtime Library*), με σκοπό τον συγχρονισμό νημάτων. Μια παραλλαγή του μηχανισμού συναντάται ακόμα και στο περιβάλλον των Windows, με την κλήση *WaitOnAddress()* [4].

Καθώς έχει προηγηθεί σημαντική έρευνα σχετική με την έξυπνη χρήση των futexes, η διπλωματική εργασία αυτή δεν πραγματεύεται τόσο την δημιουργία καινούργιων μεθόδων συγχρονισμού νημάτων, όσο την εκτενή μελέτη των αντίστοιχων μεθόδων που έχουν δημιουργηθεί και παρακάμπτουν τη χρήση της pthreads βιβλιοθήκης, χρησιμοποιώντας μόνο futexes και κώδικα χρήστη. Τέλος, στην παρούσα εργασία γίνεται λόγος για την εγκυρότητα και αποτελεσματικότητα των μεθόδων αυτών, μέσα από την εξέταση του αντίκτυπου της χρήσης τους σε ένα ολοκληρωμένο πρόγραμμα. Το πρόγραμμα που επιλέχθηκε είναι ο C-σε-C μεταφραστής ανοικτού κώδικα *OMPi* (*OpenMP C Compiler*) [5], ο οποίος παρουσιάζεται παρακάτω. Η μεθοδολογία της εξαγωγής συμπερασμάτων θα σχετίζεται με την επίδοση του κώδικα που παράγεται από τον τροποποιημένο OMPi, ο οποίος θα αντικαθιστά κάθε κλήση συγχρονισμού νημάτων pthreads με μια αντίστοιχη κλήση που χρησιμοποιεί μόνο futex και κώδικα χρήστη.

1.3.1 Σύντομη παρουσίαση του OMPi

Ο OMPi είναι ένας παραλληλοποιητικός μεταφραστής ανοιχτού κώδικα για προγράμματα που έχουν γραφτεί σε γλώσσα C και υποστηρίζει το πρότυπο OpenMP. Η έκδοση που χρησιμοποιείται στα πλαίσια της εργασίας είναι η 3.0.0. Αποτελείται

από δύο τμήματα:

1. **Τον μεταφραστή (compiler)** που δέχεται ως είσοδο προγράμματα γραμμένα σε γλώσσα προγραμματισμού C με εντολές OpenMP. Ο συντακτικός αναλυτής διατρέχει το πρόγραμμα και παράγει το συντακτικό δέντρο, το οποίο στη συνέχεια επεξεργάζεται κατάλληλα ώστε να αφαιρέσει τις οδηγίες OpenMP.

2. **Τη βιβλιοθήκη χρόνου εκτέλεσης (runtime libraries)** που παρέχει συναρτήσεις για την δημιουργία υπολογιστικών οντοτήτων (όπως νήματα και διεργασίες) που αναλαμβάνουν την εκτέλεση του κώδικα που έχει παραλληλοποιηθεί, συναρτήσεις που διαχειρίζονται κλειδαριές και άλλες βοηθητικές δυνατότητες, καθώς και συναρτήσεις που διαχειρίζονται την επικοινωνία με συσκευές και την εκτέλεση τμημάτων κώδικα σε αυτές.

Πιο συγκεκριμένα, ο μεταφραστής OMPi μετασχηματίζει πηγαίο κώδικα C που περιέχει οδηγίες OpenMP σε βελτιστοποιημένο πολυνηματικό κώδικα C. Η διαδικασία ξεκινά με συντακτική ανάλυση, η οποία περιλαμβάνει λεκτική ανάλυση και δημιουργία συντακτικού δέντρου. Στη συνέχεια, κάθε οδηγία OpenMP αντικαθίσταται με αντίστοιχες κλήσεις της βιβλιοθήκης υποστήριξης εκτέλεσης. Το αποτέλεσμα είναι πηγαίος κώδικας C που βασίζεται αποκλειστικά σε αυτή τη βιβλιοθήκη. Τελικό στάδιο αποτελεί η μεταγλώττιση του νέου κώδικα και η σύνδεσή του με τις απαραίτητες βιβλιοθήκες για την παραγωγή του εκτελέσιμου αρχείου, με τη χρήση του μεταφραστή του συστήματος (πχ. gcc στο Linux).

Στην παρούσα εργασία θα μας απασχολήσει περισσότερο το σύστημα υποστήριξης εκτέλεσης OMPi runtime [6], το οποίο συνοδεύει τον OMPi και για το οποίο ακολουθεί εκτενής ανάλυση σε επόμενο ξεχωριστό κεφάλαιο.

1.4 Δομή της Διπλωματικής εργασίας

Η διπλωματική εργασία απαρτίζεται από 7 κεφάλαια, εκ των οποίων το πρώτο αποτελεί την Εισαγωγή. Ακολουθεί μια περιγραφή των υπόλοιπων κεφαλαίων.

- **Κεφάλαιο 2**: Περιγραφή του OpenMP: Φιλοσοφία και τρόπος χρήσης.

- **Κεφάλαιο 3**: Περίληψη των υπάρχοντων μηχανισμών συγχρονισμού νημάτων της βιβλιοθήκης pthreads (mutex, condition variables), ανάλυση της ιδέας του

μηχανισμού των futexes και αποσαφήνιση των πειραματικών μηχανισμών συγχρονισμού νημάτων που προκύπτουν από τη χρήση του.

- **Κεφάλαιο 4**: Εξέταση του συστήματος υποστήριξης εκτέλεσης OMPi Runtime και περιγραφή της ενσωμάτωσης των πειραματικών μεθόδων συγχρονισμού νημάτων του Κεφαλαίου 2 στον πηγαίο κώδικα του OMPi.

- **Κεφάλαιο 5**: Περιγραφή του περιβάλλοντος εκτέλεσης των πειραμάτων και εξήγηση της μεθοδολογίας της εκτέλεσης αυτών. Παράθεση των αποτελεσμάτων και εξαγωγή συμπερασμάτων. Σύγκριση μεταξύ των μηχανισμών συγχρονισμού της pthreads και εκείνων του 3ου Κεφαλαίου, που χρησιμοποιούν futexes.

- **Κεφάλαιο 6**: Σύνοψη της Διπλωματικής Εργασίας και μελλοντικές επεκτάσεις.

Κεφαλαιο 2

OpenMP

Αντικείμενο του παρόντος κεφαλαίου αποτελεί η ανάλυση των δυνατοτήτων της βιβλιοθήκης-διεπαφής OpenMP, στο οποίο έγινε σύντομη αναφορά στο προηγούμενο κεφάλαιο και το οποίο μας απασχολεί στην παρούσα εργασία. Η φιλοσοφία του μεταφραστή OMPi στοχεύει στη μετάφραση προγραμμάτων σε C που χρησιμοποιούν κλήσεις του OpenMP API, οι οποίες αναφέρονται παρακάτω στο παρόν κεφάλαιο. Ξεκινάμε με μια γενική περιγραφή των προδιαγραφών του και των συστατικών στοιχείων από τα οποία αποτελείται το OpenMP.

2.1 Εισαγωγή

Σαν επανάληψη όσων αναφέρθηκαν στην εισαγωγή της εργασίας, τα νήματα POSIX με την NPTL υλοποίηση αποτελούν ίσως τον πιο διαδεδομένο τρόπο προγραμματισμού στο μοντέλο του κοινόχρηστου χώρου διευθύνσεων. Βέβαια, η χρήση τους παραμένει επίσης αρκετά πολύπλοκη και απαιτεί σημαντική εμπειρία. Ο προγραμματιστής πρέπει να χειρίζεται όλες τις λεπτομέρειες σχετικά με τη δημιουργία,

εκτέλεση, δρομολόγηση και τερματισμό των νημάτων κι αν υπάρχει ήδη σειριακός κώδικας, χρειάζεται να ξαναγραφεί και να οργανωθεί εκ νέου ώστε να υποστηρίζει πολυνηματική εκτέλεση.

Το πρότυπο προγραμματισμού OpenMP προσφέρει μια διεπαφή που αυτοματοποιεί πολλές από τις διαδικασίες και τις τεχνικές του παράλληλου προγραμματισμού, οι οποίες συνήθως υλοποιούνται χειροκίνητα από τον προγραμματιστή. Αν και ο χειρισμός της προσπέλασης των δεδομένων παραμένει σε σημαντικό βαθμό ευθύνη του προγραμματιστή, πλέον γίνεται με πιο οργανωμένο τρόπο χάρη στα εργαλεία που παρέχει το OpenMP. Γενικά, το OpenMP αποτελείται από τρία βασικά στοιχεία που είναι άμεσα προσβάσιμα στον προγραμματιστή:

1. Οδηγίες προς τον μεταγλωττιστή (pragmas ή directives), που καθορίζουν πώς θα πρέπει να διαχειριστεί συγκεκριμένα τμήματα του κώδικα ή των δεδομένων.

2. Συναρτήσεις βιβλιοθήκης, οι οποίες επιτρέπουν τη ρύθμιση παραμέτρων εκτέλεσης, τη διευκόλυνση του συγχρονισμού των νημάτων (π.χ. με χρήση locks) και την παροχή εργαλείων όπως η μέτρηση επιδόσεων. Ένα παράδειγμα είναι ο καθορισμός του αριθμού των νημάτων για την εκτέλεση ενός συγκεκριμένου τμήματος κώδικα.

3. Μεταβλητές περιβάλλοντος, που ορίζουν προκαθορισμένες παραμέτρους εκτέλεσης πριν από την έναρξη του προγράμματος.

Ο βασικός στόχος του OpenMP είναι να διευκολύνει τη σταδιακή παραλληλοποίηση ενός σειριακού προγράμματος, επιτρέποντας στον προγραμματιστή να επιταχύνει επιμέρους τμήματα του κώδικα χωρίς να αλλάξει τη βασική λογική της εφαρμογής. Επίσης, το OpenMP είναι σχεδιασμένο με γνώμονα τη φορητότητα, αφού η διαχείριση των νημάτων και των δεδομένων υλοποιείται από τον μεταγλωττιστή και όχι από τον ίδιο τον προγραμματιστή.

Η διεπαφή προγραμματισμού OpenMP καλύπτει μόνο την παραλληλοποίηση που καθοδηγείται από τον χρήστη, δηλαδή ο προγραμματιστής καθορίζει ρητά τις ενέργειες που πρέπει να εκτελέσει ο μεταγλωττιστής και το σύστημα χρόνου εκτέλεσης για να τρέξει το πρόγραμμα παράλληλα. Οι υλοποιήσεις που συμμορφώνονται με το πρότυπο OpenMP δεν υποχρεούνται να ελέγχουν για εξαρτήσεις δεδομένων, συγκρούσεις δεδομένων, αγώνες δεδομένων (data races) ή αδιέξοδα (deadlocks).

Επίσης, δεν απαιτείται να ανιχνεύουν τμήματα κώδικα που θα καθιστούσαν το πρόγραμμα μη συμμορφούμενο με το πρότυπο. Η ευθύνη για τη σωστή χρήση της διεπαφής OpenMP και τη δημιουργία ενός συμμορφούμενου προγράμματος βαρύνει τον προγραμματιστή. Επιπλέον, η διεπαφή OpenMP δεν περιλαμβάνει αυτόματη παραλληλοποίηση, η οποία πραγματοποιείται από τον μεταγλωττιστή.

Το OpenMP παρέχει πλήθος ισχυρών προγραμματιστικών εργαλείων που απλοποιούν τη διαδικασία δημιουργίας παράλληλων προγραμμάτων και βοηθούν τον προγραμματιστή να αξιοποιήσει στο μέγιστο τις επιδόσεις του παράλληλου συστήματος όπου εκτελείται η εφαρμογή. Η ανάλυση ξεκινάει με μια περιγραφή του μοντέλου εκτέλεσης (Execution Model) του OpenMP.

2.2 Μοντέλο Εκτέλεσης του OpenMP

Η διεπαφή OpenMP χρησιμοποιεί το μοντέλο εκτέλεσης fork-join, όπου πολλαπλά νήματα εκτελούν εργασίες που ορίζονται από τις οδηγίες OpenMP. Το OpenMP υποστηρίζει προγράμματα που μπορούν να τρέξουν σωστά είτε παράλληλα (με πολλά νήματα) είτε σειριακά (όταν οι οδηγίες αγνοούνται). Ένα πρόγραμμα OpenMP ξεκινά με ένα αρχικό-κύριο νήμα (master thread) που εκτελεί τον κώδικα σειριακά, σαν να βρίσκεται σε μια αρχική, ''έμμεση'' παράλληλη περιοχή (implicit parallel region). Για κάθε περιοχή παραλληλίας (parallel region) που συναντάται, δημιουργείται (fork) μία ομάδα νημάτων, στην οποία θα συμμετέχει και το κύριο νήμα. Η ομάδα νημάτων θα εκτελέσει παράλληλα το τμήμα κώδικα που περικλείεται από την οδηγία παραλληλοποίησης και με το πέρας της περιοχής παραλληλίας, η ομάδα νημάτων θα καταστραφεί (join) και η εκτέλεση του υπόλοιπου θα συνεχιστεί ξανά από το κύριο νήμα.

Η εκτέλεση γίνεται κυρίως στη συσκευή υποδοχής (host), η οποία μπορεί να αναθέτει εκτέλεση κώδικα και δεδομένων σε άλλες συσκευές στόχους (target devices), κάθε μία με τα δικά της νήματα. Τα νήματα δεν μεταφέρονται μεταξύ συσκευών, και το μοντέλο εκτέλεσης είναι προσανατολισμένο στη συσκευή υποδοχής [7]. Εν τέλει, ο τρόπος με τον οποίο ο προγραμματιστής εκμεταλλεύεται το μοντέλο αυτό εξαρτάται πάντα κυρίως από την σειρά και το είδος των οδηγιών OpenMP που εισαγάγει ο χρήστης σε ένα πρόγραμμα και οι οποίες παρουσιάζονται παρακάτω.

2.3 Οδηγίες OpenMP για C/C++

Η υποενότητα παρουσιάζει βασικές λειτουργίες του OpenMP για τις γλώσσες C και C++, καλύπτοντας τη σύνταξη οδηγίας, τη δημιουργία περιοχών παραλληλίας, τον διαμοιρασμό εργασίας μεταξύ νημάτων, τις οδηγίες συγχρονισμού και σημαντικές παραμέτρους (clauses) που ρυθμίζουν την εκτέλεση περιοχών OpenMP.

2.3.1 Σύνταξη

Οι οδηγίες OpenMP στις γλώσσες C/C++ ορίζονται με χρήση της οδηγίας προεπεξεργαστή *pragma*. Η γενική μορφή είναι:

#pragma omp όνομα_οδηγίας [φράση[, φράση] ...]

και ακολουθεί νέα γραμμή. Ισχύουν οι εξής κανόνες:

1. Το όνομα της οδηγίας είναι υποχρεωτικό μετά το #pragma omp.
2. Το όνομα της οδηγίας είναι case-sensitive.

Μετά την οδηγία, μπορούν προαιρετικά να τοποθετηθούν φράσεις, που καθορίζουν τις συνθήκες εκτέλεσης, χωρίς περιορισμό στη σειρά τους. Αν δεν υπάρχουν φράσεις, οι συνθήκες ορίζονται κατά την εκτέλεση. Η οδηγία πρέπει να τελειώνει με νέα γραμμή.

2.3.2 Περιοχές Παραλληλίας (Parallel Regions)

Ως περιοχή παραλληλίας ορίζεται το τμήμα κώδικα που εκτελείται παράλληλα από μία ομάδα νημάτων. Η περιοχή δηλώνεται μέσω της οδηγίας *parallel* και περιλαμβάνει ένα δομημένο τμήμα, που μπορεί να είναι είτε μπλοκ κώδικα μέσα σε άγκιστρα είτε μία μοναδική εντολή χωρίς άγκιστρα. Τρόπος χρήσης της parallel:

#pragma omp parallel [φράση[[,] φράση] ...] νέα-γραμμή
δομημένο τμήμα

Κατά την εκτέλεση, το κύριο νήμα δημιουργεί την ομάδα νημάτων που εκτελεί το δομημένο τμήμα. Τα νήματα μπορεί να τερματίζουν σε διαφορετικούς χρόνους, γι' αυτό στο τέλος της περιοχής υπάρχει ένα φράγμα (barrier) που συγχρονίζει τα νήματα, ώστε όλα να ολοκληρώσουν πριν συνεχίσει το κύριο πρόγραμμα. Ο χρήστης μπορεί να καθορίσει το μέγεθος της ομάδας νημάτων με τρεις βασικές μεθόδους:

1. Κλήση της συνάρτησης **omp_set_num_threads()** εκτός περιοχής παραλληλίας.

2. Ορισμός της μεταβλητής περιβάλλοντος **OMP_NUM_THREADS** πριν την εκτέλεση.

3. Χρήση της φράσης `num_threads()` στην οδηγία parallel με τον αριθμό των νημάτων που επιθυμεί.

2.3.3 Διαμοιρασμός Εργασίας (Worksharing Regions)

Ο διαμοιρασμός της εργασίας στα νήματα μιας ομάδας αποτελεί βασική λειτουργία του OpenMP, επιτρέποντας την αποτελεσματική κατανομή του φόρτου εργασίας, όπως για παράδειγμα σε επαναληπτικούς βρόχους μέσα σε μια περιοχή παραλληλίας. Για το σκοπό αυτό, το OpenMP υποστηρίζει τις περιοχές διαμοιρασμού εργασίας, που παρέχουν στον προγραμματιστή ειδικές οδηγίες για να κατανείμει το υπολογιστικό φορτίο σε όλα τα νήματα της ομάδας. Αν και αυτές οι περιοχές μπορούν να χρησιμοποιηθούν και εκτός περιοχών παραλληλίας, η χρησιμότητά τους είναι μεγαλύτερη όταν εφαρμόζονται μέσα σε αυτές.

Όπως και στις περιοχές παραλληλίας, στο τέλος κάθε περιοχής διαμοιρασμού εργασίας υπάρχει ενσωματωμένο φράγμα (barrier) που συγχρονίζει τα νήματα, διασφαλίζοντας ότι όλα ολοκληρώνουν πριν συνεχιστεί η εκτέλεση. Ωστόσο, η χρήση της φράσης nowait στην οδηγία επιτρέπει την παράλειψη του φράγματος, προσφέροντας ευελιξία και βελτιστοποίηση σε κατάλληλες περιπτώσεις.

Υπάρχουν τρεις βασικοί τύποι περιοχών διαμοιρασμού εργασίας στο OpenMP, καθένας ορισμένος από συγκεκριμένες οδηγίες, που βοηθούν στην ορθή και αποδοτική κατανομή του φόρτου σε παράλληλα εκτελούμενα τμήματα του κώδικα.

1. **Οδηγία `for`**: Χρησιμοποιείται για τον καταμερισμό του φόρτου ενός επαναληπτικού βρόχου `for`. Ακολουθεί ο βρόχος που παραλληλοποιείται. Σύνταξη:

   #pragma omp for [φράση[, φράση] ...]
   for (αρχικοποίηση; συνθήκη; βήμα) {
   ...
   }

2. **Οδηγία `sections`**: Επιτρέπει την παραλληλοποίηση ανεξάρτητων εργασιών εντός περιοχής παραλληλίας. Περιλαμβάνει διαδοχικές δηλώσεις `#pragma omp section` με δομημένα τμήματα κώδικα. Σύνταξη:

   #pragma omp sections [φράση[, φράση] ...]
   #pragma omp section
   δομημένο τμήμα

3. **Οδηγία `single`**: Επιτρέπει την εκτέλεση ενός δομημένου τμήματος μόνο από το πρώτο νήμα που το συναντά. Τα υπόλοιπα νήματα περιμένουν έως την ολοκλήρωση, για συγχρονισμό. Παραλλαγή: `master`, όπου το δομημένο τμήμα εκτελείται μόνο από το κύριο νήμα και δεν υπονοείται φράγμα συγχρονισμού. Σύνταξη:

   #pragma omp single | master [φράση[, φράση] ...]
   δομημένο τμήμα

2.3.4 Συγχρονισμός Νημάτων στο OpenMP

Οι οδηγίες συγχρονισμού στο OpenMP εξασφαλίζουν τη σωστή και συνεπή εκτέλεση των νημάτων κατά την παράλληλη επεξεργασία. Οι βασικότερες είναι οι εξής:

1. **Οδηγία `atomic`**: διασφαλίζει ότι μια πράξη σε μια μεταβλητή εκτελείται αδιαίρετα, χωρίς παρέμβαση άλλων νημάτων.

2. **Οδηγία `barrier`**: φράγμα όπου όλα τα νήματα περιμένουν μέχρι να φτάσουν όλα, ώστε να συνεχίσουν ταυτόχρονα με ενημερωμένες κοινόχρηστες μεταβλητές.

3. **Οδηγία `critical`**: ορίζει κρίσιμη περιοχή κώδικα που εκτελείται μόνο από ένα νήμα κάθε φορά, με τα υπόλοιπα να περιμένουν.

4. **Οδηγία `flush`**: συγχρονίζει τη μνήμη μεταξύ νημάτων, εξασφαλίζοντας ότι όλες οι αλλαγές σε κοινόχρηστα δεδομένα είναι ορατές σε όλα τα νήματα.

5. **Οδηγία `ordered`**: Η οδηγία ordered χρησιμοποιείται για την σειριοποίηση της εκτέλεσης ενός τμήματος κώδικα εντός μιας παράλληλης περιοχής. Συνήθως τοποθετείται στο εσωτερικό ενός βρόχου επανάληψης, όταν χρειάζεται να τηρηθεί συγκεκριμένη σειρά στην εκτέλεση των επαναλήψεων.

Αυτές οι οδηγίες είναι απαραίτητες για την αποφυγή συγκρούσεων και την ορθή συνεργασία των νημάτων στο παράλληλο περιβάλλον.

2.3.5 Φράσεις Οδηγιών

Οι φράσεις οδηγιών στο OpenMP τοποθετούνται μετά το όνομα της οδηγίας και καθορίζουν τον τρόπο εκτέλεσής της, καθώς και τη συμπεριφορά των νημάτων πριν, κατά τη διάρκεια και μετά το δομημένο τμήμα κώδικα. Βασικές φράσεις είναι:

1. `if(`**συνθήκη**`)`: Εκτελεί την οδηγία μόνο αν η συνθήκη είναι αληθής.

2. `shared/private/firstprivate/lastprivate` **(λίστα μεταβλητών)**: Ορίζουν τον τρόπο ορατότητας και ζωής των μεταβλητών στην παράλληλη περιοχή, όπως κοινόχρηστες ή ιδιωτικές με αρχικές ή τελικές τιμές.

3. `nowait`: Αποφεύγει το φράγμα συγχρονισμού που υπονοείται μετά την οδηγία, επιτρέποντας στα νήματα να συνεχίσουν χωρίς αναμονή.

4. `num_threads()`: Καθορίζει τον αριθμό των νημάτων που θα χρησιμοποιηθούν σε μια παράλληλη περιοχή.

5. `schedule(`**τύπος**`[,` **μέγεθος κόκκου**`])`: Ρυθμίζει τον τρόπο κατανομής επαναλήψεων βρόχου στα νήματα, με πολιτικές όπως static (στατικός καταμερισμός), dynamic (δυναμικός) και guided (με μειούμενο μέγεθος τμημάτων). Υπάρχουν τρεις πολιτικές διαμοιρασμού των επαναλήψεων του βρόχου στους επεξεργαστές ή/και πυρήνες: η **static** που διαμοιράζει στατικά τον βρόχο σε τμήματα σταθερού μεγέθους ή ανά νήμα, η `dynamic` όπου τα διαθέσιμα νήματα αναλαμβάνουν δυναμικά τμήματα εργασίας, και η **guided** που λειτουργεί παρόμοια με τη dynamic αλλά μειώνει εκθετικά το μέγεθος των τμημάτων κατά την εκτέλεση.

6. `reduction(`**πράξη** : **λίστα μεταβλητών**`)`: Επιτρέπει την εκτέλεση αθροιστικών πράξεων σε κοινόχρηστες μεταβλητές, εξασφαλίζοντας τη συνέπεια των δεδομένων χωρίς επιπλέον κώδικα από τον προγραμματιστή.

Αυτές οι φράσεις προσφέρουν ευελιξία και έλεγχο στην παράλληλη εκτέλεση.

2.3.6 Tasks

Η οδηγία `task`, εισαχθείσα στην έκδοση 3.0 του OpenMP, επιτρέπει την εκτέλεση παράλληλου κώδικα με δυναμικό τρόπο, χωρίς τον περιορισμό του στατικού τρόπου παραλληλισμού. Μέσω αυτής, ο προγραμματιστής ορίζει τμήματα κώδικα (tasks) που μπορούν να εκτελεστούν οποιαδήποτε στιγμή υπάρχει διαθέσιμος επεξεργαστής ή παράλληλα με άλλες περιοχές παραλληλισμού. Αυτός ο τύπος παραλληλισμού είναι ιδανικός για εφαρμογές όπου η ισορροπία φόρτου μεταξύ νημάτων είναι κρίσιμη, καθώς επιτρέπει την εναλλαγή της εκτέλεσης ενός task μεταξύ νημάτων, δηλαδή ένα task μπορεί να ξεκινήσει από ένα νήμα και να συνεχιστεί από άλλο. Ο προγραμματιστής μπορεί να καθορίσει τις μεταβλητές του task με φράσεις όπως `private`, `shared` και `firstprivate`, για σωστή λειτουργία. Για τον συγχρονισμό των tasks χρησιμοποιούνται οι οδηγίες `barrier`, που έχει συζητηθεί, και `taskwait`, που αναστέλλει την εκτέλεση ενός task μέχρι να ολοκληρωθούν τα υπο-tasks που έχει δημιουργήσει.

2.3.7 Συσκευές (OpenMP Devices)

Από την έκδοση 4.0 του OpenMP και μετά, δίνεται η δυνατότητα εκτέλεσης κώδικα σε συσκευές εκτός της CPU, όπως συνεπεξεργαστές, επιταχυντές και κάρτες γραφικών. Η CPU λειτουργεί ως "host" και η εξωτερική συσκευή ως "device". Η οδηγία **#pragma omp `target`** επιτρέπει την αποστολή κώδικα στη συσκευή, ενώ με τις φράσεις `to`, `from`, `tofrom` και `alloc`, ορίζεται η κατεύθυνση και ο τρόπος μεταφοράς των δεδομένων. Η `to` χρησιμοποιείται για αρχικοποίηση μεταβλητών στη συσκευή, η `from` για επιστροφή αποτελεσμάτων στον `host`, η `tofrom` για αμφίδρομη μεταφορά, και η `alloc` για δέσμευση χώρου χωρίς μεταφορά. Για πρόσβαση σε καθολικές μεταβλητές ή συναρτήσεις στη συσκευή, απαιτείται η χρήση των περιοχών `declare`/`end declare`. Με την έκδοση 4.5 προστέθηκαν επιπλέον δυνατότητες διαχείρισης και εκτέλεσης σε συσκευές, ενισχύοντας τη λειτουργικότητα πέρα από την αρχική υποστήριξη.

2.3.8 Συναρτήσεις Βιβλιοθήκης του OpenMP

Το OpenMP παρέχει μια σειρά από συναρτήσεις μέσω της βιβλιοθήκης χρόνου εκτέλεσης (omp.h), τις οποίες μπορεί να αξιοποιήσει ο προγραμματιστής. Οι συ-

ναρτήσεις αυτές επιτρέπουν, μεταξύ άλλων, τον δυναμικό καθορισμό του αριθμού των νημάτων ανάλογα με τους διαθέσιμους πόρους του συστήματος, τον συγχρονισμό των νημάτων με τη χρήση μηχανισμών όπως οι κλειδαριές (locks), καθώς και την άντληση πληροφοριών από το σύστημα, όπως το πλήθος των επεξεργαστών. Επιπλέον, περιλαμβάνονται συναρτήσεις για χρονομέτρηση και άλλες βοηθητικές λειτουργίες. Μεταξύ αυτών, ορισμένες θεωρούνται βασικές για την ανάπτυξη και διαχείριση παράλληλων εφαρμογών. Μερικές βασικές συναρτήσεις, όπως η **omp_set_num_threads()** που συναντήσαμε νωρίτερα στο κείμενο, αποτελούν οι παρακάτω:

1. **omp_get_num_threads()**: Επιστρέφει το πλήθος των νημάτων που χρησιμοποιούνται.

2. **omp_set_num_threads()**: Ορίζει το πλήθος των νημάτων που θα εκτελέσουν μια παράλληλη περιοχή (υπερισχύει της μεταβλητής περιβάλλοντος **OMP_NUM_THREADS**).

3. **omp_get_thread_num()**: Επιστρέφει τον αριθμό του τρέχοντος νήματος (μεταξύ 0 και ((πλήθος νημάτων) -1)).

4. **omp_get_num_procs()**: Επιστρέφει το πλήθος των διαθέσιμων πυρήνων που βλέπει το σύστημα.

5. **omp_init_lock()**: Αρχικοποιεί μια απλή κλειδαριά.

6. **omp_destroy_lock()**: Αφαιρεί μια κλειδαριά.

7. **omp_set_lock()**: Το νήμα περιμένει να γίνει διαθέσιμη η κλειδαριά.

8. **omp_unset_lock()**: Παραχωρεί τη διαθεσιμότητα της κλειδαριάς.

9. **omp_get_wtime()**: Επιστρέφει το χρόνο που έχει περάσει από κάποια στιγμή στο παρελθόν (δεν αλλάζει κατά το χρόνο εκτέλεσης) και χρησιμοποιείται για χρονομετρήσεις.

10. **omp_set_nested()**: Ενεργοποιεί nested παραλληλισμό (εξετάζεται αργότερα στο κείμενο).

2.3.9 Μεταβλητές Περιβάλλοντος του OpenMP

Σε συστήματα UNIX και λειτουργικά συστήματα Linux, το OpenMP επιτρέπει τον έλεγχο της εκτέλεσης μέσω μεταβλητών περιβάλλοντος. Η πιο χαρακτηριστική είναι η **OMP_NUM_THREADS**, η οποία καθορίζει πόσα νήματα θα δημιουργηθούν, εκτός αν δηλώνονται μέσα στο πρόγραμμα. Η **OMP_NESTED** ενεργοποιεί ή απενεργοποιεί τον ένθετο παραλληλισμό, ενώ η **OMP_SCHEDULE** επηρεάζει τη συμπεριφορά της παραμέτρου schedule στην οποία αναφερθήκαμε νωρίτερα. Άλλες μεταβλητές σχετίζονται με επιδόσεις, συσκευές και πολιτικές διαχείρισης νημάτων.

Μια ιδιαίτερα σημαντική ομάδα μεταβλητών αφορά την τοποθέτηση των νημάτων στους επεξεργαστικούς πυρήνες. Αυτή η ομάδα μεταβλητών θα μας απασχολήσει στον τρόπο διεξαγωγής των πειραμάτων και θα εξηγηθεί κι εκεί. Για την ώρα, αναφέρουμε πως η **OMP_PLACES** καθορίζει τους διαθέσιμους επεξεργαστικούς πόρους (π.χ. threads, cores, sockets) και επιτρέπει και προσαρμοσμένες ομαδοποιήσεις. Η **OMP_PROC_BIND** ελέγχει τη δέσμευση νημάτων στους πόρους: με spread, τα νήματα διασπείρονται ομοιόμορφα· με close, τοποθετούνται κοντά στο master νήμα· και με master, εκτελούνται στο ίδιο place με αυτό. Για παράδειγμα, σε σύστημα με δύο sockets και 16 πυρήνες, **OMP_PLACES**=`sockets` διανέμει τα νήματα εναλλάξ ανά socket, ενώ **OMP_PLACES**=`cores` με **OMP_PROC_BIND**=`close` τοποθετεί διαδοχικά τα νήματα σε πυρήνες του ίδιου socket πριν προχωρήσει στον επόμενο. Αυτές οι μεταβλητές επιτρέπουν βελτιστοποίηση της απόδοσης και προβλέψιμη συμπεριφορά κατά την εκτέλεση.

Κεφαλαιο 3

Mutexes, Futexes και Συγχρονισμος Νηματων

Έχοντας εξηγήσει τις προηγμένες λειτουργίες που προσφέρει το OpenMP, μπορούμε να προχωρήσουμε στις λεπτομέρειες χρήσης και της ιδέας των μεθόδων συγχρονισμού νημάτων που προσφέρονται από την pthreads, την καθιερωμένη βιβλιοθήκη διαχείρισης των νημάτων, που έχει συζητηθεί μόνο περιληπτικά μέχρι τώρα. Μόνο μετά από την εν λόγω ανάλυση θα είμαστε σε θέση να αναλωθούμε στην μελέτη των futexes και τον τρόπο που μπορούμε να αντικαταστήσουμε τις κλήσεις του pthreads με ισοδύναμες (και ίσως πιο αποδοτικές) που χρησιμοποιούν futexes. Επιπλέον, η συνολική προεργασία είναι απαραίτητη ώστε να έχει δημιουργηθεί το απαραίτητο υπόβαθρο και στο επόμενο Κεφάλαιο να γίνει σαφής ο τρόπος λειτουργίας του συστήματος υποστήριξης εκτέλεσης του OMPi, αλλά και των αλλαγών που

προτείνονται για την χρήση των μηχανισμών συγχρονισμού με futexes και κώδικα χρήστη, που αναφέρονται παρακάτω στο παρόν Κεφάλαιο.

3.1 Τρόποι Συγχρονισμού Νημάτων με pthreads

Ο προγραμματιστής που επιθυμεί να χρησιμοποιήσει την pthreads (pthreads.h) ώστε να δρομολογήσει την σωστή εκτέλεση των νημάτων ενός προγράμματος, μπορεί να βασιστεί στην χρήση των μεθόδων που προφέρονται και χωρίζονται σε δύο βασικές κατηγορίες: Κλήσεις που υλοποιούν Mutex(Αμοιβαίο αποκλεισμό, Mutual Exclusion) και κλήσεις που υλοποιούν Condition Variables (Μεταβλητές Συνθήκης. Άλλοι τρόποι συγχρονισμού αποτελούν το φράγμα ή barrier, μέσω των κλήσεων `pthread_barrier_init()`, `pthread_barrier_destroy()` και `pthread_barrier_wait()` και των λουκέτων ανάγνωσης-εγγραφής (read-write locks ή rwlocks) μέσω των κλήσεων `pthread_rwlock_init()`/ `pthread_rwlock_destroy()`, `pthread_rwlock_rdlock()`/ `pthread_rwlock_wrlock()` και `pthread_rwlock_unlock()`. Θα αρκεστούμε μόνο στα επεξηγηματικά ονόματα των κλήσεων που αναφέρθηκαν και θα επικεντρωθούμε στις κλήσεις που αφορούν τον αμοιβαίο αποκλεισμό με απλές `pthread_mutex` κλειδαριές και τις μεταβλητές συνθήκης.

3.1.1 Mutex στο pthreads

Ένα mutex λέμε πως χρησιμοποιείται για να προστατεύει κρίσιμες περιοχές κώδικα, στις οποίες γίνεται κοινοχρησία πόρων κι άρα επικρατούν συνθήκες ανταγωνισμού. Στην κλασσική περίπτωση, όταν ένα νήμα θέλει να εισέλθει σε μια τέτοια περιοχή, προσπαθεί πρώτα να ”κλειδώσει το αντίστοιχο mutex”. Αν το mutex είναι ελεύθερο, το νήμα εισέρχεται και το mutex ”κλειδώνεται”, εμποδίζοντας άλλα νήματα. Αν είναι ήδη κλειδωμένο, το νήμα μπλοκάρεται μέχρι να απελευθερωθεί και τελικά να αφυπνηθεί το νήμα που προσπάθησε να το κλειδώσει. Αν υπάρχουν πολλά νήματα που περιμένουν, μόνο ένα συνεχίζει όταν το mutex ξεκλειδωθεί.

Η σωστή χρήση των mutexes με τη χρήση της pthreads επαφίεται πράγματι στον προγραμματιστή, ο οποίος φροντίζει για την σωστή κλήση στην σωστή σειρά προγράμματος. Οι βασικές σχετικές συναρτήσεις είναι οι εξής:

1. **pthread_mutex_init(pthread_mutex_t *mutex) και**

pthread_mutex_destroy(pthread_mutex_t *mutex) για την δημιουργία και καταστροφή μιας κλειδαριάς

2. **phread_mutex_lock(pthread_mutex_t *mutex)** για την απόπειρα κλειδώματος που είναι εμποδιστική (blocking). Δηλαδή, η κλήση επιτυγχάνει στο κλείδωμα μιας ελεύθερης/μη-κατειλημμένης κλειδαριάς από το καλών νήμα και αποτυγχάνει στο κλείδωμα μιας κατειλημμένης κλειδαριάς και οδηγεί στο μπλοκάρισμα του καλούντος νήματος

3. **pthread_mutex_trylock(pthread_mutex_t *mutex)** για μη-εμποδιστικό (non-blocking) έλεγχο κατάστασης ενός mutex και απόπειρας κλειδώματος αυτού

4. **pthread_mutex_unlock(pthread_mutex_t *mutex)** για ξεκλείδωμα της κλειδαριάς

Τα mutexes μπορούν επίσης να έχουν χαρακτηριστικά (attributes) για εξειδικευμένες περιπτώσεις. Οι λειτουργίες αυτές επιτρέπουν ασφαλή πρόσβαση σε κοινά δεδομένα από πολλαπλά νήματα. Αξίζει να αναφερθεί πως υπάρχει και εναλλακτικός τρόπος αρχικοποίησης ενός mutex, μέσω στατικής αρχικοποίησης. Άλλωστε, ένα ''mutex'' αποτελεί και μια μεταβλητή τύπου `pthread_mutex_t`, η οποία μπορεί να αρχικοποιηθεί ως `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;` στην αρχή ενός αρχείου κώδικα. Η στατική αρχικοποίηση είναι χρήσιμη για την ρύθμιση της ορατότητας μιας μεταβλητής. Αν για παράδειγμα επιθυμούμε μια μεταβλητή mutex να είναι ορατή σε όλο το πρόγραμμα, χρησιμοποιούμε την στατική αρχικοποίηση.

Επιπλέον, η pthreads υποστηρίζει τη χρήση `spin locks` με τον τύπο δεδομένων `pthread_spinlock_t` και τις αντίστοιχες κλήσεις όπως `pthread_spin_lock()`, `pthread_spin_unlock()`. Αυτού του τύπου οι κλειδαριές χρησιμοποιούν την τεχνική του busy-waiting αντί του μπλοκαρίσματος του καλούντος νήματος, που μπορεί να φανεί χρήσιμη σε συγκεκριμένες περιπτώσεις (πχ. όταν η κρίσιμη περιοχή εμπεριέχει μικρό αριθμό εντολών).

Τέλος, η υποστήριξη ένθετου παραλληλισμού επιτυγχάνεται έμμεσα μέσω της δυναμικής αρχικοποίησης. Σε αυτήν την περίπτωση, με τη χρήση πιο προχωρημένων κλήσεων της pthreads ειδοποιούμε το σύστημα πως η κλειδαριά τύπου `pthread_mutex_t *mutex` που θέλουμε να χρησιμοποιήσουμε ανήκει στην κατηγορία `PTHREAD_MUTEX_RECURSIVE` και ενώ χρησιμοποιείται η γνωστή `pthread_mutex_lock()` για το κλείδωμα, το ξε-

κλείδωμα μέσω της `pthread_mutex_unlock()` επιτυγχάνει μόνο όταν το καλών νήμα έχει "ξεκλειδώσει την κλειδαριά" ακριβώς όσες φορές την έχει "κλειδώσει". Με αυτού του τύπου τις κλειδαριές (*recursive*) επιτυγχάνεται ο συγχρονισμός των νημάτων εντός περιοχών ένθετου παραλληλισμού, στις οποίες υπάρχουν περιοχές παράλληλης εκτέλεσης (πχ. όσες σηματοδοτούνται με `#pragma omp parallel` στον OpenMP) εντός τέτοιων περιοχών.

3.1.2 Condition Variables στο pthreads

Ενώ τα mutexes είναι κατάλληλα για την επίτρεψη ή την απαγόρευση πρόσβασης σε μια κρίσιμη περιοχή, οι μεταβλητές συνθήκης (condition variables) επιτρέπουν στα νήματα να μπλοκάρονται όταν δεν ισχύει κάποια συγκεκριμένη συνθήκη.

Οι βασικές λειτουργίες-κλήσεις των condition variables είναι οι εξής:

1. **pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)** και **pthread_cond_signal(pthread_cond_t *cond)**. Η `pthread_cond_wait()` ξεκλειδώνει ένα `mutex` που λαμβάνει σαν όρισμα κι αμέσως μπλοκάρει το νήμα που την καλεί μέχρι κάποιο άλλο νήμα να το ειδοποιήσει με την κλήση `pthread_cond_signal()`. Ο λόγος αναμονής δεν αποτελεί μέρος του ίδιου του μηχανισμού, αλλά σχετίζεται με εξωτερικές ενέργειες, όπως η απελευθέρωση πόρων ή η ολοκλήρωση εργασιών από άλλα νήματα.

2. Η **pthread_cond_broadcast(pthread_cond_t *cond)** χρησιμοποιείται όταν πολλά νήματα περιμένουν στην ίδια μεταβλητή συνθήκης, ώστε να ξυπνήσουν όλα ταυτόχρονα.

3. Η **pthread_cond_init(pthread_cond_t *cond)** χρησιμοποιείται για την αρχικοποίηση μιας μεταβλητής συνθήκης και η **pthread_cond_destroy(pthread_cond_t *cond)** για την καταστροφή της και την αποδέσμευση της μνήμης που κατελάμβανε.

Σε πλήρη αντιστοιχία με τα `mutexes`, έτσι και ο όρος condition variable ή μεταβλητή συνθήκης αποτελεί μια μεταβλητή τύπου `pthread_cond_t` και επιτρέπεται η στατική της αρχικοποίηση, πέραν της δυναμικής μέσω της `pthread_cond_init()` ως εξής, στην αρχή ενός αρχείου κώδικα: `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`.

3.2 Τι είναι futex

Η ιδέα της χρήσης των futex βασίζεται σε μια έξυπνη παρατήρηση: όταν ένα νήμα συναντήσει μια ελεύθερη κλειδαριά, το κλείδωμά της μπορεί να είναι "φθηνό" και μπορεί να πραγματοποιηθεί κλείδωμα (και ξεκλείδωμα) χωρίς μια "ακριβή" κλήση συστήματος, με την εκτέλεση πολύ "φθηνότερων" **ατομικών λειτουργιών** [1]. Ατομικές ονομάζονται οι λειτουργίες που θεωρούνται αδιαίρετες και χωρίς τη δυνατότητα διακοπής από κάποιο σήμα, λόγω της απευθείας επικοινωνίας τους με το υλικό του συστήματος. Για αυτό, μια ατομική λειτουργία είτε εκτελείται τελείως είτε καθόλου.

Πληροφοριακά αναφέρουμε πως κάθε λειτουργία του `futex` μπορεί να αφορά και νήματα αλλά και διεργασίες, μόνο που στη δεύτερη περίπτωση απαιτείται δημιουργία περιοχής κοινόχρηστης μνήμης (πχ. με `mmap()` ή `shmat()`). Αν δεν γίνει δημιουργία περιοχής κοινόχρηστης μνήμης των διεργασιών, τότε δε θα υπάρχει τρόπος να ελέγξει μια διεργασία την κατάσταση της κλειδαριάς. Η ανάλυση της περίπτωσης συγχρονισμού διεργασιών με τη χρήση `futexes` ξεφεύγει από τους σκοπούς της εργασίας αυτής.

Στην περίπτωση σύμφωνα με την οποία ένα άλλο νήμα προσπαθήσει να πάρει το mutex την ίδια στιγμή, η προσέγγιση με ατομικές λειτουργίες μπορεί να αποτύχει. Σε αυτή την περίπτωση υπάρχουν δύο επιλογές.

1. Μπορούμε να εκτελέσουμε busy-loop χρησιμοποιώντας μια ατομική εντολή μέχρι να αφεθεί η κλειδαριά και χωρίς να χρειαστούμε τον πυρήνα (kernel). Μπορεί να αποτελέσει εξαιρετικά σπάταλη επιλογή, αφού ο βρόχος μπορεί να μονοπωλήσει έναν πυρήνα και το κλείδωμα μπορεί να κρατηθεί για μεγάλο χρονικό διάστημα.

2. Η εναλλακτική λύση είναι να μπλοκάρουμε/κοιμήσουμε το νήμα μέχρι η κλειδαριά να ελευθερωθεί και να αφυπνισθεί το νήμα - χρειαζόμαστε τον πυρήνα να βοηθήσει σε αυτό, και εδώ είναι που χρησιμοποιούνται τα futexes.

Προτού δείξουμε την κλήση συστήματος για το futex, πρέπει να θυμόμαστε πως η υλοποίηση των futexes δεν συνοδεύεται από λειτουργίες κλειδώματος/ξεκλειδώματος. Αντ' αυτού είναι αποτελούν "εργαλείο" ή *primitive* που μπορεί να χρησιμοποιηθεί για την υλοποίηση μεθόδων συγχρονισμού στο χώρο των χρηστών (`userspace`). Οι

δύο σχετικές και πιο βασικές λειτουργίες ονομάζονται `FUTEX_WAIT` και `FUTEX_WAKE`. Αναλύονται παρακάτω στο ίδιο κεφάλαιο.

Το άλλο σημαντικό γεγονός που έχουμε αναφέρει, είναι πως ενώ ο όρος "`futex`" μπορεί να αποτελεί κλήση συστήματος, αποτελεί ταυτόχρονα και το όνομα μιας ακέραιας μεταβλητής 32-bit - που συναντάται στην βιβλιογραφία κι ως *futex word* - της οποίας η διεύθυνση πρέπει να παρέχεται στην κλήση συστήματος που αντιστοιχεί στο `futex`. Το μέγεθος ενός `futex-word` είναι 32 bits σε όλες τις πλατφόρμες, συμπεριλαμβανομένων των συστημάτων 64-bit. Όλες οι λειτουργίες futex εξαρτώνται από αυτή την μεταβλητή, της οποίας καμιά τιμή δεν έχει μια συγκεκριμένη σημασία.

Καθώς η βιβλιοθήκη της C (*GNU C Library* ή `glibc`) δεν περιλαμβάνει συνάρτηση wrapper για το futex, ακολουθεί η κλήση συστήματος (μέσω `syscall()`) που αντιστοιχεί στο `futex`, όπως παρουσιάζεται στην ομόνυμη σελίδα στο manpage του Linux [2].

```
#include <linux/futex.h>      /* Definition of FUTEX_* constants */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_futex, uint32_t *uaddr, int futex_op, uint32_t val,
             const struct timespec *timeout,/* or: uint32_t val2 */
             uint32_t *uaddr2, uint32_t val3);
```

Κώδικας 3.1: Κλήση συστήματος για το futex

Ακολουθεί εξήγηση των πιο βασικών ορισμάτων, τα οποία μας φαίνονται χρήσιμα για την ανάλυσή μας και χρησιμοποιούνται συχνά στα πλαίσια της εργασίας αυτής:

- **uaddr**: Δείκτης που δείχνει το futex-word. Σε όλα τα συστήματα, τα futex είναι ακέραιοι αριθμοί τεσσάρων byte που πρέπει να ευθυγραμμίζονται σε ένα όριο τεσσάρων byte. Για εξοικονόμηση χώρου κι ακρίβειας, χρησιμοποιείται ο τύπος μη-προσημασμένου ακεραίου `uint32_t`.

- **futex_op**: Η λειτουργία που θα εκτελεστεί με τη κλήση συστήματος του futex καθορίζεται από την τιμή αυτού του ορίσματος.

- **val**: Αποτελεί μια τιμή της οποίας το νόημα και ο σκοπός εξαρτάται από το `futex_op`.

Τα υπόλοιπα ορίσματα (`timeout`, `uaddr2` και `val3`) απαιτούνται μόνο για ορισμένες από τις λειτουργίες που δεν αφορούν την παρούσα μελέτη κι όταν ένα από αυτά τα ορίσματα δεν απαιτείται για μια λειτουργία, αγνοείται. Συγκεκριμένα, για αρκετές εμποδιστικές λειτουργίες (blocking operations πχ. κοίμηση), το όρισμα `timeout` είναι ένας δείκτης σε μια δομή `timespec`, που καθορίζει ένα χρονικό όριο για μια λειτουργία (πχ. κοίμηση για ένα χρονικό διάστημα). Όπου απαιτείται, το όρισμα `uaddr2` είναι ένας δείκτης σε μια δεύτερη `futex word` που χρησιμοποιείται αναλόγως από το σύστημα. Η ερμηνεία του τελευταίου ακέραιου ορίσματος, `val3`, εξαρτάται από την λειτουργία και δεν χρησιμοποιείται από όλες τις διαθέσιμες.

Για πληρότητα αναφέρουμε πως η κλήση συστήματος για το futex επιστρέφει ακέραιο αριθμό τύπου long int και η σημασία του ερμηνεύεται ανάλογα με την λειτουργία που εκτελεί η εκάστοτε κλήση futex και την έκβασή της (πχ. αν πέτυχε ή απέτυχε).

Στην εργασία αυτή χρησιμοποιείται η παρακάτω συνάρτηση wrapper, που έχει ήδη συμπληρωμένες τις τιμές των ορισμάτων που δεν μας είναι χρήσιμα.

```
1 int futex(uint32_t *uaddr, int futex_op, uint32_t val)
2 {
3     return syscall(SYS_futex, uaddr, futex_op, val, NULL, NULL, 0);
4 }
```

Κώδικας 3.2: Συνάρτηση-περιέκτης (wrapper) του futex

Είναι καιρός να αναλύσουμε τις δύο πιο δημοφιλείς λειτουργίες των futexes και οι οποίες μας αφορούν για τους σκοπούς μας. Στην ουσία, τα παρακάτω ονόματα των λειτουργιών χρησιμοποιούνται αυτούσια για τον ορισμό του ορίσματος futex_op.

- **FUTEX_WAIT**: Με αυτή η λειτουργία γίνεται έλεγχος της τιμής του futex word που υποδεικνύεται από τη διεύθυνση uaddr. Αν την στιγμή του ελέγχου περιέχει την αναμενόμενη τιμή val συμβαίνουν τα εξής: Αν ναι, τότε το καλών νήμα κοιμίζεται κι αναμένει ένα σήμα αφύπνισης που προκύπτει από μια λειτουργία FUTEX_WAKE. Η φόρτωση της τιμής της futex word αποτελεί μια ατομική προσπέλαση μνήμης (δηλ. χρησιμοποιώντας εντολές μηχανής της αντίστοιχης αρχιτεκτονικής επεξεργαστή). Εάν το νήμα είναι κοιμησμένο, λέμε ότι παρακολουθεί ή "περιμένει" (*waits on the value*) σε αυτή τη futex word. Εάν η τιμή που είναι αποθηκευμένη στο futex word δεν ισούται με την val, τότε η κλήση αποτυγχάνει αμέσως κι επιστρέφεται το σφάλμα EAGAIN. Επιστρέφει 0 εάν ο καλών έχει ξυπνήσει. Στην περίπτωση στην οποία το νήμα δεν κοιμίζεται επειδή η τιμή του futex word δεν ισούται με την τιμή val, επιστρέφει κωδικό σφάλματος EAGAIN ή EWOULDBLOCK (αντιστοιχούν στην ίδια ακέραια τιμή).

- **FUTEX_WAKE**: Αυτή η λειτουργία αφυπνίζει το πολύ val νήματα (ή διεργασίες) από όσα περιμένουν τη futex word στη διεύθυνση uaddr. Συνήθως, η val καθορίζεται είτε ως 1 (αφύπνιση ενός μόνο "κοιμισμένου") είτε ως INT_MAX (αφύπνιση όλων των "κοιμισμένων"). Δεν παρέχεται καμία εγγύηση σχετικά

με το ποια νήματα αφυπνίζονται και δεν τηρείται κάποια προτεραιότητα. Εγγύηση παρέχεται μόνο όσον αφορά το πλήθος τους. Η κλήση επιστρέφει τον αριθμό των νημάτων που ξύπνησαν.

3.3 Χρήση των futexes

Για να κατανοήσουμε πλήρως τις δυνατότητες των futexes, στο παράρτημα κώδικα της εργασίας παρατίθεται ένα παράδειγμα υπολογισμού του αριθμού π με χρήση πολλαπλών νημάτων, κλειδαριών με futex και OpenMP.

```
#include <stdio.h>
// Error reporting
#include <errno.h>
// Futex lib
#include <linux/futex.h>
// OpenMP parallelism
#include <omp.h>
// Other dependencies needed to use malloc etc
#include <sys/syscall.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
// Define LIMIT for loop iterations
#define LIMIT 100000000

// Wrapper for futex syscall
static int futex(uint32_t *uaddr, int futex_op,
                 uint32_t val, const struct timespec *timeout,
                 uint32_t *uaddr2, uint32_t val3) {
    return syscall(SYS_futex, uaddr, futex_op, val, timeout, uaddr2,
       val3);
}
// Futex lock
void flock(uint32_t *futexp){
    while (1){
        if (__sync_val_compare_and_swap(futexp, 1, 0)) // If the
```

```
            lock is available, make it unavailable and break
26              break;
27          futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
28      }
29      return;
30  }
31  // Futex unlock
32  void funlock(uint32_t *futexp){
33      (void)__sync_fetch_and_add(futexp, 1);
34      futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
35      return;
36  }
37  int main(int argc, char *argv[])
38  {
39      uint32_t *futex_lock = (uint32_t *)malloc(sizeof(uint32_t));
40      *futex_lock = 1; // lock available
41      double PI = 0, W = 1.0/LIMIT;
42      #pragma omp parallel
43      {
44          int i;
45          double temp = 0.0;
46          #pragma omp for
47          for (i=0;i<LIMIT;i++)
48              temp += 4*W / (1+(i+0.5)*(i+0.5)*W*W);
49          flock(futex_lock);
50          //#pragma omp critical
51          PI += temp;
52          funlock(futex_lock);
53      }
54      free(futex_lock);
55      printf("%0.10lf\n", PI);
56      return 0;
57  }
```

Κώδικας 3.3: Υπολογισμός του PI με τη χρήση συναρτήσεων κλειδώματος/ξεκλειδώματος με futex

Η εκτέλεση του παραπάνω προγράμματος εμφανίζει στην οθόνη την στρογγυλοποίηση του π στα 10 δεκαδικά ψηφία (3.1415926536). Ο σωστός υπολογισμός προκύπτει από την μετατροπή του ολοκληρώματος

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

σε μερικά αθροίσματα

$$S_{\text{partial}} = \sum_{i=i_{\text{start}}}^{i_{\text{end}}} \frac{4W}{1+((i+0.5)W)^2}$$

και την ανάθεση του υπολογισμού κάθε αθροίσματος σε ένα νήμα και μόνο, ο οποίος εκτελείται παράλληλα με κάποιον άλλον. Κάθε νήμα προσθέτει το άθροισμα που υπολογίζει στη μεταβλητή στην οποία συσσωρεύεται το τελικό αποτέλεσμα, το οποίο θα πρέπει να ισούται με το π.

Παρόλο που με την πρώτη ματιά φαίνεται πως ο κώδικας εκτελεί 0 έως LIMIT-1 επαναλήψεις, στην πραγματικότητα το OpenMP διαμοιράζει τις επαναλήψεις σε έναν αυθαίρετο αριθμό νημάτων και κάθε νήμα διαθέτει ένα ιδιωτικό αντίγραφο της μεταβλητής i, εφόσον υπάρχει η οδηγία `#pragma omp parallel` και η οδηγία `#pragma omp for` είναι εμφωλευμένη.

Όσον αφορά τον συγχρονισμό νημάτων, η χρήση κλειδαριών αρκεί και παρακάμπτεται η οδηγία `#pragma omp critical` για τον ορισμό της κρίσιμης περιοχής. Στην κρίσιμη περιοχή τροποποιείται η κοινόχρηστη μεταβλητή PI και η πρόσβασή της ελέγχεται με τη βοήθεια των λειτουργιών κλειδώματος-ξεκλειδώματος. Ας εστιάσουμε την προσοχή μας στη λειτουργία κλειδώματος:

```
1 void flock(uint32_t *futexp){
2     while (1){
3         if (__sync_val_compare_and_swap(futexp, 1, 0)) // If the
              lock is available, make it unavailable and break
4             break;
5         futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
6     }
7     return;
8 }
```

Κώδικας 3.4: flock

Το σκεπτικό της παρούσας συνάρτησης είναι να ελέγξει την κατάσταση της κλειδαριάς: αν είναι κλειδωμένη τότε το καλών νήμα πρέπει να περιμένει. Στην αντίθετη περίπτωση, χρησιμοποιεί την κλειδαριά και αποκτά πρόσβαση στην κρίσιμη περιοχή που ακολουθεί. Μετά, πρέπει υποχρεωτικά να πραγματοποιήσει και άφεση της κλειδαριάς ώστε κάποιο επόμενο νήμα να κλειδώσει την κλειδαριά. Χρησιμοποιούμε την τιμή του δείκτη `futexp` ώστε να συμβολίζουμε την τρέχουσα κατάσταση της κλειδαριάς. Καταχρηστικά θα αναφερόμαστε στον δείκτη ως "κλειδαριά".

Έστω πως αν η τιμή στην οποία δείχνει ισούται με 1 η κλειδαριά είναι διαθέσιμη και αν ισούται με 0 τότε δεν είναι. Στην περίπτωση της συνάρτησης υπο εξέταση, ο έλεγχος γίνεται μέσω της ατομικής CAS (Compare And Store) συνάρτησης `__sync_val_compare_and_swap (futexp , 1, 0)`, η οποία με τα εμφανιζόμενα ορίσματα αναθέτει την τιμή 0 στην `futexp` αν και μόνο αν η τιμή της είναι 1 την στιγμή που καλείται. Η τιμή που επιστρέφεται είναι η τιμή της futexp τη στιγμή που καλείται.

Στην περίπτωση που υπάρχει ανταγωνισμός και η κλειδαριά έχει καταληφθεί, τότε η σύγκριση της εντολής `__sync_val_compare_and_swap (futexp , 1, 0)` αποτυγχάνει, επιστρέφεται 0, παρακάμπτεται η εντολή `break` και η ροή του προγράμματος προχωράει στην κλήση της `futex()` συνάρτησης. Σύμφωνα με τα ορίσματα με τα οποία εμφανίζεται στην συνάρτηση `flock()` και σύμφωνα με την προηγούμενη ανάλυση, η εκτέλεση του νήματος "μπλοκάρεται" μέχρι να ξυπνήσει από μια κλήση τύπου `FUTEX_WAKE`. Με άλλα λόγια σε αυτή τη γραμμή σταματά η εκτέλεση του νήματος αν η κλειδαριά είναι κατειλημμένη.

Δύο παρατηρήσεις:

1. Όλες οι εντολές της `flock()` εμπεριέχονται σε έναν βρόχο ώστε να αποφευχθεί η περίπτωση να αλλάξει η κατάσταση της κλειδαριάς μετά τον έλεγχό της και πριν την κλήση `futex()` για μπλοκάρισμα. Αν αλλάξει, λογικό είναι να επιθυμούμε να επανεξεταστεί η κατάσταση της κλειδαριάς και να επαναληφθεί η διεκδίκησή της.

2. Για τον ίδιο λόγο, εύκολα μπορούμε να παρατηρήσουμε πως αν η εξέταση της τιμής εκτελούνταν μέσω μιας κλασσικής if-else δομής, θα υπήρχε το ενδεχόμενο σύμφωνα με το οποίο ένα άλλο νήμα θα μπορούσε να αλλάξει την κατάσταση της κλειδαριάς πριν την αλλάξει το καλών νήμα. Έτσι, δύο νήματα θα μπορούσαν να αποκτήσουν πρόσβαση στην κρίσιμη περιοχή.

Ας εξετάσουμε την λειτουργία του ξεκλειδώματος:

```
1 void funlock(uint32_t *futexp){
2     (void)__sync_fetch_and_add(futexp, 1);
3     futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
4     return;
5 }
```

Κώδικας 3.5: funlock

Η `funlock()` αποτελεί σαφώς μικρότερη συνάρτηση σε έκταση, αλλά παραμένει αποτελεσματική. Εκτελεί δύο ενέργειες: Αλλάζει την κατάσταση της κλειδαριάς σε "ελεύθερη" μέσω ατομικής λειτουργίας και προκαλεί την αφύπνιση ενός νήματος, μέσω της κλήσης της `futex(futexp , FUTEX_WAKE , 1, NULL , NULL , 0)`, η οποία με τη σειρά της καλεί το σύστημα να αφυπνίσει ένα νήμα από εκείνα που αναμένουν μπλοκαρισμένα την κλειδαριά. Σαφώς υποθέτουμε πως πριν την κλήση της συνάρτησης `funlock()` έχει προηγηθεί η κλήση της `flock()` από το ίδιο νήμα.

3.4 Βελτιωμένη υλοποίηση κλειδαριάς με τη χρήση futexes

Έχοντας κατανοήσει τη χρήση των futexes σε ένα βασικό σενάριο με μια απλή υλοποίηση, μπορούμε να παρατηρήσουμε τα εξής:

- Η υλοποίηση της `funlock()` προκαλεί μια κλήση συστήματος κάθε φορά που καλείται, που πάντα θεωρείται ακριβή. Ακόμα κι αν έχει δημιουργηθεί μόνο ένα νήμα για να εκτελέσει τον κώδικα, η `funlock()` πάντα καλείται και προκαλεί την ακριβή κλήση συστήματος.

- Έστω πως υπάρχουν τρία νήματα με ταυτότητες Α, Β και Γ αντίστοιχα και εκτελούν τον παραπάνω κώδικα του παραδείγματος. Ένα πιθανό σενάριο σειράς εκτέλεσης είναι το εξής: Το νήμα Α καλεί πρώτο την συνάρτηση κλειδώματος και ανακαλύπτει πως η κλειδαριά είναι ελεύθερη κι άρα θέτει `*futexp = 0`. Το νήμα Β ελέγχει την κατάσταση της κλειδαριάς και ανακαλύπτει πως είναι κατειλημμένη, η τιμή του δείκτη `*futexp` ισούται με 0 ακόμα και την στιγμή που καλείται η `FUTEX_WAIT` και οπότε η εκτέλεσή του μπλοκάρεται , μέχρι που συνεχίζει με την αφύπνισή του από την κλήση της `funlock()` από το νήμα Α. Ωστόσο, αν το νήμα Γ ανακαλύψει πως η κλειδαριά είναι ελεύθερη

αλλά η εκτέλεση της `funlock()` έχει προηγηθεί της λειτουργίας `FUTEX_WAIT`, τότε το νήμα Γ δεν μπλοκάρεται (επιστρέφεται `EWOULDBLOCK`) και τελικά επιτυγχάνει τον έλεγχο της κλειδαριάς.

Το πρόβλημα και στις δύο περιπτώσεις είναι το γεγονός ότι δεν υπάρχει κάποια πρόβλεψη όσον αφορά τον αριθμό των νημάτων που αναμένουν την απόκτηση του ελέγχου της κλειδαριάς. Για παράδειγμα, στην δεύτερη παρατήρηση βλέπουμε πως το νήμα Γ αποκτά πρόσβαση στην κρίσιμη περιοχή χωρίς να μπλοκαριστεί κι άρα να απαιτηθεί η αφύπνισή του. Παράλληλα όμως, παρατηρούμε πως η έννοια του αμοιβαίου αποκλεισμού δεν παραβιάστηκε εφόσον το νήμα Α και Β εκτέλεσαν κανονικά τις εργασίες τους χωρίς παρεμβολές και διακοπές.

Γενικά, ο προγραμματιστής μιας υλοποίησης κλειδαριάς πρέπει να είναι ενήμερος για τα παρακάτω γεγονότα:

1. Το πολύ ένα νήμα ανά πάσα στιγμή μπορεί να κατέχει τη κλειδαριά (να έχει κλειδώσει την κλειδαρία, δηλαδή να είναι ο *ιδιοκτήτης*) και

2. Εάν η κλειδαριά είναι ελεύθερο τότε είτε ένα είτε περισσότερα νήματα πάντα προσπαθούν να αποκτήσουν τον έλεγχο της κλειδαριάς, εφόσον η ουρά των αναμένοντων νημάτων δεν είναι άδεια.

Η αλήθεια είναι πως με την κοίμηση νήματος μέσω του μηχανισμού των `futexes`, το νήμα τοποθετείται σε μια κατακερματισμένη ουρά της οποίας τη διαχείριση αναλαμβάνει ο πυρήνας. Κάτι που πρέπει να έχουμε στο νου μας σε μια υλοποίηση κλειδαριάς με τη χρήση futexes, είναι πως ο μηχανισμός των futexes δεν εγγυάται ύπαρξη δικαιοσύνης όσον αφορά ποιο νήμα θα αφυπνηθεί. Ωστόσο, αυτό το γεγονός αλλάζει καθώς μπορούμε να χρησιμοποιήσουμε πιο προηγμένες λειτουργίες που προσφέρουν τα `futexes` και να κατασκευάσουμε *Priority Inheritance (PI) futexes*. Σε αυτήν την περίπτωση, το `futex` δεν σχετίζεται μόνο με την κατάσταση της κλειδαριάς, αλλά και με την ταυτότητα του νήματος - ιδιοκτήτη της κλειδαριάς και του αν υπάρχουν κι άλλα νήματα τα οποία αναμένουν την απόκτηση της κλειδαριάς. Έτσι, το σύστημα γνωρίζει ποιο νήμα έχει προτεραιότητα κι αποφεύγεται το ενδεχόμενο αναμονής σημαντικού νήματος λόγω εργασίας ενός λιγότερου σημαντικού νήματος (βλ. *Priority Inversion Problem*). Αυτή η περίπτωση δεν εμπεριέχεται στους σκοπούς μελέτης της εργασίας και αρκεί να υποθέτουμε πως στην συνηθισμένη περίπτωση που μελετάται, η επιλογή του νήματος προς αφύπνιση από το σύστημα γίνεται τυχαία.

Για την ικανοποίηση των παραπάνω συνθηκών, μπορούμε να χρησιμοποιήσουμε μια διαφορετική υλοποίηση [8] σύμφωνα με την οποία κάθε κλειδαριά μπορεί να βρίσκεται σε μία από τρείς καταστάσεις αντί για δύο:

- **0**: Διαθέσιμη κλειδαριά
- **1**: Μη διαθέσιμη κλειδαριά και δεν υπάρχει κανένα άλλο νήμα να αναμένει.
- **2**: Μη διαθέσιμη κλειδαριά και ένα ή περισσότερα νήματα αναμένουν την κλειδαριά.

Για λόγους που σχετίζονται με την συνέπεια μνήμης και θα εξηγηθούν αργότερα στο ίδιο κεφάλαιο, παραθέτουμε μια συνάρτηση wrapper μιας πιο σύγχρονης συνάρτησης CAS που προσφέρει ο μεταφραστής gcc:

```
1 int cmpxchg(uint32_t *ptr, int expected, int desired){
2     return __atomic_compare_exchange_n(ptr, &expected, desired, 0,
          __ATOMIC_ACQ_REL, __ATOMIC_RELAXED);
3 }
```

Κώδικας 3.6: Η συνάρτηση cmpxchg (Compare And Exchange)

Στη γενική περίπτωση, η `__atomic_compare_exchange_n(type *ptr, type *expected, type desired, bool weak, int success_memorder, int failure_memorder)` συγκρίνει τα περιεχόμενα του `*ptr` με τα περιεχόμενα του `*expected`. Εάν είναι ίσα μεταξύ τους, τότε η λειτουργία είναι μια λειτουργία ανάγνωσης-τροποποίησης-εγγραφής που αποθηκεύει την τιμή `desired` στο `*ptr`. Αν δεν είναι ίσα, η λειτουργία είναι μια ανάγνωση και τα τρέχοντα περιεχόμενα του `*ptr` γράφονται στο `*expected`. Από αυτό το σημείο της εργασίας κι έπειτα, χρησιμοποιούνται οι `__atomic` εντολές για την εκτέλεση των ατομικών λειτουργιών. Το όνομα κάθε μιας είναι αρκετά επεξηγηματικό και φανερώνει εύκολα τον ρόλο κάθε εντολής.

Στην `__atomic_compare_exchange_n()`, εάν η επιθυμητή τιμή τελικά αποθηκεύεται στο `*ptr`, τότε επιστρέφεται true (1) και η μνήμη τροποποιείται σύμφωνα με τη συνέπεια μνήμης που καθορίζεται από την `success_memorder`. Στην αντίθετη περίπτωση, επιστρέφεται false (0) και ισχύει η `failure_memorder`.

Εστιάζουμε την προσοχή μας στην καινούργια `flock()`:

```
1 void flock(uint32_t *futexp)
2 {
```

```
3     if (!(cmpxchg(futexp,0,1))){
4         do{
5             if (__atomic_load_n(futexp, __ATOMIC_ACQUIRE) == 2 ||
                  cmpxchg(futexp,1,2))
6                  futex(futexp, FUTEX_WAIT, 2);
7         }while(!cmpxchg(futexp,0,2));
8     }
9 }
```

Κώδικας 3.7: Καινούργια υλοποίηση της flock()

Έστω πως ένα νήμα διεκδικεί μια κλειδαρία που βρίσκεται σε ένα futex. Κοιτώντας τον κώδικα και λαμβάνοντας υπόψιν όλα τα παραπάνω, η `flock()` λειτουργεί ως εξής:

- **Η κλειδαριά είναι διαθέσιμη**: Σε αυτή την περίπτωση, πετυχαίνει η εναλλαγή της τιμής της κλειδαριάς με την `cmpxchg()` και το νήμα αποκτά τον έλεγχο της κλειδαριάς, η συνθήκη στην οποία εμπεριέχεται η κλήση αυτή αποτιμάται ως `false` κι έτσι ολοκληρώνεται η εκτέλεση της συνάρτησης.

- **Η κλειδαριά δεν είναι διαθέσιμη**: Τότε, υπάρχουν δύο υπό-περιπτώσεις. Τις εξετάζουμε ξεχωριστά:

    1. Αν η τιμή της κλειδαριάς ισούται με 1 την στιγμή που το νήμα καλεί την `flock()`, τότε κανένα άλλο νήμα δεν έχει προσπαθήσει να τη διεκδικήσει. Εντός του `do-while` βρόχου αποφασίζεται αν το νήμα θα μπλοκαριστεί. Τότε, υπάρχει η αισιόδοξη περίπτωση η οποία προβλέπει πως το άλλο νήμα που κατέχει την κλειδαριά θα έχει ολοκληρώσει την εργασία του και θα θέσει την τιμή της σε 0. Τότε, ο έλεγχος της γραμμής 5 της `flock()` θα αποτύχει και το νήμα θα προσπαθήσει να κλειδώσει την κλειδαριά θέτοντας την τιμή 2 εφόσον υπήρξε διεκδίκηση. Αντιθέτως, στην απαισιόδοξη περίπτωση, ο έλεγχος της γραμμής 5 της `flock()` πετυχαίνει καθώς είτε άλλο νήμα προσπάθησε ταυτόχρονα να διεκδικήσει την κλειδαριά είτε επειδή το νήμα - ιδιοκτήτης δεν έχει ολοκληρώσει την εργασία του. Και στις δύο περιπτώσεις αυτές, η τιμή της κλειδαριάς είτε ισούται με 2 είτε της ανατίθεται η τιμή 2 και το νήμα που διεκδικεί την κλειδαριά μπλοκάρεται.

2. Αν η τιμή της κλειδαριάς ισούται με 2 την στιγμή που το νήμα καλεί την `flock()`, τότε υπάρχει ξανά η αισιόδοξη περίπτωση το νήμα - ιδιοκτήτης της κλειδαριάς να ολοκληρώσει την εργασία του, να αλλάξει έγκαιρα η τιμή της κλειαριάς και να μην χρειαστεί να μπλοκαριστεί το νήμα που καλεί την `flock()`. Στην αντίστοιχη απαισιόδοξη περίπτωση, το νήμα κοιμίζεται καθώς η τιμή της κλειδαριάς ισούται με 2 και ο έλεγχος της γραμμής 5 εντός του βρόχου `do-while` πετυχαίνει.

Σημαντική παρατήρηση αποτελεί το γεγονός ότι στην γραμμή 5, στον έλεγχο που διενεργείται μέσω της `if`, ο έλεγχος `c==2` βρίσκεται στο αριστερό μέρος της πράξης `or` που πραγματοποιείται. Αυτό επειδή ο έλεγχος μιας `or` στην `C/C++` γίνεται από αριστερά προς τα δεξιά και σταματάει τη στιγμή που βρεθεί μια αληθής υποσυνθήκη. Αν λοιπόν τη κλειδαριά διεκδικείται από τουλάχιστον ένα νήμα, υπάρχει μια γρήγορη παράκαμψη στον κώδικα η οποία οδηγεί στην κοίμηση του νήματος.

Τώρα, εστιάζουμε την προσοχή μας στην καινούργια `funlock()`:

```
1 void funlock(uint32_t *futexp)
2 {
3     if (__atomic_fetch_sub(futexp, 1, __ATOMIC_ACQ_REL) != 1) {
4         __atomic_store_n (futexp, 0, __ATOMIC_RELEASE);
5         futex(futexp, FUTEX_WAKE, 1);
6     }
7 }
```

Κώδικας 3.8: Καινούργια υλοποίηση της funlock()

Σε αντίθεση με την προηγούμενη υλοποίηση, στην καινούργια υπάρχει τέτοια μέριμνα που απαγορεύει την κλήση συστήματος `FUTEX_WAKE` στην περίπτωση όπου δεν υπάρχουν νήματα στην ουρά αναμονής. Για αυτόν τον λόγο, μέσω την εντολής `if` της γραμμής 3, ελέγχουμε ποιο είναι το αποτέλεσμα της αφαίρεσης (μέσω ατομικής λειτουργίας) της μονάδας (το 1) από την τιμή της κλειδαριάς `futex`. Οι δύο τιμές που συγκρίνονται μεταξύ τους την στιμή της κλήσης της `funlock()` είναι η παλιά τιμή της μεταβλητής της κλειδαριάς πριν την αφαίρεση και της μονάδας (το 1) και η διαφορά που περιγράφτηκε αποθηκεύεται στην μεταβλητή της κλειδαριάς (στην τιμή που δείχνει ο δείκτης `futexp`). Προκύπτουν οι εξής περιπτώσεις:

- **Το αποτέλεσμα της αφαίρεσης είναι 0**: Τότε, η παλιά τιμή της κλειδαριάς είναι 1 κι άρα δεν υπήρχε κανένα νήμα στην ουρά αναμονής. Σε αυτήν την

περίπτωση, η διαδικασία κλειδώματος-ξεκλειδώματος έχει πραγματοποιηθεί αμιγώς ατομικά, χωρίς καμιά κλήση συστήματος.

- **Το αποτέλεσμα της αφαίρεσης είναι 1**: Τότε, η παλιά τιμή της κλειδαριάς είναι 2 κι άρα υπάρχει τουλάχιστον ένα νήμα στην ουρά αναμονής. Οπότε, για να αφυπνιστεί κάποιο από αυτά ώστε να κλειδώσει την κλειδαριά, θέτουμε την κλειδαριά στην κατάσταση διαθεσιμότητας (γραμμή 4) κι εκτελείται `FUTEX_WAKE`.

Το ενδεχόμενο η διαφορά να ισούται με -1 αγνοείται καθώς υποθέτουμε πως πριν μια κλήση της συνάρτησης ξεκλειδώματος υπάρχει πάντα μια κλήση συνάρτησης κλειδώματος `flock()`, από το ίδιο νήμα. Επιπροσθέτως, και στις δύο υλοποιήσεις συνάρτησης ξεκλειδώματος φροντίζουμε να αφυπνίζουμε μόνο ένα νήμα με την εισαγωγή του αριθμού 1 στην κλήση `FUTEX_WAKE`. Αυτό το γεγονός αποτελεί βέλτιστη πρακτική που αποτρέπει την εμφάνιση του `Thundering Herd` προβλήματος, σύμφωνα με το οποίο πολλά νήματα αφυπνίζονται μαζικά και ανταγωνίζονται αμέσως για την απόκτηση ενός κοινόχρηστου πόρου. Το γεγονός αυτό θα μπορούσε να προκαλέσει μεγάλη σπατάλη κύκλων CPU και ανεπιθύμητες καθυστερήσεις.

3.5 Υλοποίηση μηχανισμού μεταβλητών συνθήκης με τη χρήση futexes

Όπως αναφέρθηκε νωρίτερα στο παρόν κείμενο, ο μηχανισμός αμοιβαίου αποκλεισμού συνδυάζεται με τη χρήση μεταβλητών συνθήκης. Σαφώς είναι σκόπιμο να κατασκευάσουμε έναν ο οποίος επίσης χρησιμοποιεί αποκλειστικά `futexes` και κώδικα χρήστη. Άλλωστε, οι μέθοδοι μεταβλητών συνθήκης απαιτούν εξ ορισμού την αφύπνιση (**signal**) και αναμονή/κοίμηση (**wait**) νημάτων ή διεργασιών - λειτουργίες που προσφέρει η χρήση των `futexes`.

Για την ακρίβεια, ο προγραμματιστής μιας υλοποίησης ενός μηχανισμού μεταβλητών συνθήκης (condition variables) πρέπει να είναι ενήμερος για τον βασικό ορισμό των δύο απαραίτητων μεθόδων που απαιτούνται:

1. Η λειτουργία της `wait()` απαιτεί το ξεκλείδωμα μιας κλειδαριάς της οποίας ιδιοκτήτης πρέπει να είναι το καλών νήμα, την κοίμηση του καλούντος νήματος

και την προσπάθεια κλειδώματος της ίδιας κλειδαριάς, αφότου γίνει η αφύπνισή του. Για αυτούς τους λόγους η `wait()` απαιτεί δύο ορίσματα. Το ένα όρισμα αποτελεί την μεταβλητή συνθήκης (πχ. τύπου `pthreads_cond_t`) και το άλλο την κλειδαριά (πχ. τύπου `pthreads_mutex_t`).

2. Η λειτουργία της `signal()` απαιτεί ένα όρισμα που αποτελεί την μεταβλητή συνθήκης και εκτελεί την δημιουργία ενός σήματος αφύπνισης το οποίο προορίζεται αποκλειστικά για ένα νήμα. Το νήμα αυτό θα είναι εκείνο το οποίο εκτέλεσε `wait()` χρησιμοποιώντας ως όρισμα μεταβλητής συνθήκης το ίδιο με αυτό που καλείται η `signal()`.

Ιδανικά, οι δύο συναρτήσεις πρέπει να επιστρέφουν κάποια τιμή στην περίπτωση της επιτυχούς κοίμησης/αφύπνισης (πχ. 0) και κάποιες άλλες τιμές ανάλογα με το σφάλμα που μπορεί να προκύψει. Κατ'επέκταση, ακολουθώντας το πρότυπο POSIX θα μπορούσαμε να εξετάσουμε και τη δημιουργία μιας `broadcast()` συνάρτησης για την αφύπνιση όλων των νημάτων που έκαναν κλήση της `wait()` με την ίδια μεταβλητή συνθήκης - δεν είναι χρήσιμη στην παρούσα μελέτη.

Βάσει των παραπάνω, μπορούμε να καταλήξουμε στις εξής υλοποιήσεις συνάρτησης `wait()` και `signal()` [9]. Τις ονομάζουμε `fwait()` και `fsignal()` αντίστοιχα.

```
void fwait(uint32_t *condp, uint32_t *mutexp) {
    __atomic_store_n(condp, 1, __ATOMIC_RELEASE);
    funlock (mutexp);
    futex(condp, FUTEX_WAIT, 1);
    flock(mutexp);
}
```

Κώδικας 3.9: Υλοποίηση της fwait()

```
void fsignal(uint32_t *condp) {
    __atomic_store_n(condp, 0, __ATOMIC_RELEASE);
    futex(condp, FUTEX_WAKE, 1);
}
```

Κώδικας 3.10: Υλοποίηση της fsignal()

Λόγω της απλότητας των συναρτήσεων, μπορούμε να τις δούμε συνολικά: Όταν ένα νήμα χρειαστεί να περιμένει για κάποιον λόγο καλώντας την `fwait()`, εκτελεί τα βήματα του ορισμού που δόθηκε νωρίτερα. Τα αντίστοιχα βήματα εκτελούνται για την αφύπνιση μέσω της εκτέλεσης της `fsignal()` από κάποιο άλλο νήμα. Δυστυχώς, αν και υπάρχουν περιθώρια βελτίωσης, δεν είναι αρκετά σημαντικά για τους σκοπούς της παρούσας εργασίας.

Σημαντικό ωστόσο είναι να παρατηρήσουμε την χρήση της ακέραιας μεταβλητής συνθήκης - ακέραιου δείκτη `condp` και πώς η τιμή της μεταβάλλεται μεταξύ των κλήσεων: στην `fwait()` η τιμή της τίθεται 1 και κατόπιν η `condp` χρησιμοποείται στην κλήση της `FUTEX_WAIT`. Στην κλήση αυτή όμως και σύμφωνα με τον ορισμό της λειτουργίας της `FUTEX_WAIT` που προσφέρουν τα `futexes`, πραγματοποιείται μια σύγκριση της τιμής της `condp` και της τιμής 1. Ακολούθως, η ισότητα των τιμών μεταξύ τους σημαίνει πως δεν προηγήθηκε κανένα σήμα αφύπνισης και πως το νήμα μπορεί να μπλοκαριστεί με ασφάλεια. Στην αντίθετη περίπτωση όμως, κάποιο άλλο νήμα έχει καλέσει την `fsignal()` και η τιμή της `condp` έχει μεταβληθεί και ισούται με 0. Τότε, αποτυγχάνει η σύγκριση που περιγράφτηκε, το νήμα δεν κοιμίζεται και διεκδικεί την κλειδαριά. Γενικά, στόχος μιας σωστής υλοποίησης μεθόδων μεταβλητών συνθήκης πρέπει να είναι η αποφυγή και η σωστή διαχείριση των *χαμένων σημάτων αφύπνισης* (*Lost Wakeups*).

3.6 Αναμενόμενες βελτιώσεις

Πριν η ανάλυση προχωρήσει στον τρόπο ενσωμάτωσης των παραπάνω μεθόδων στον OMPi, μπορούμε να κάνουμε μια μικρή βολιδοσκόπηση των πιθανών βελτιώσεων στην επίδοση των μεταφρασμένων προγραμμάτων που θα προκύψουν από τον τροποποιημένο OMPi. Με άλλα λόγια, μπορούμε να προσπαθήσουμε να απαντήσουμε στο ερώτημα του τι βελτιώσεις προκύπτουν στην επίδοση ενός παράλληλου προγράμματος που χρησιμοποιεί τις παραπάνω μεθόδους αντί για αυτές που προσφέρει η pthreads ή/και το OpenMP.

Όσον αφορά και τις δύο βιβλιοθήκες, αρκεί να θυμόμαστε πως η υλοποίησή τους δεν στοχεύει μόνο στις επιδόσεις αλλά και στην ορθότητα, εκτός από την απλότητα της χρήσης τους. Για παράδειγμα, ένας άλλος τύπος κλειδαριών POSIX αποτελούν τα λεγόμενα `spinlocks`, των οποίων ο ορισμός θα μας απασχολήσει στο επόμενο κεφάλαιο. Ωστόσο, αυτό που μας αφορά σε αυτό το σημείο είναι το γεγονός ότι ένας επεξεργαστής θα πρέπει να αφιερώσει μερικούς κύκλους εντολών ώστε να εκτελέσει τις εντολές απόφασης του κώδικα της `pthreads_mutex_lock()` σχετικά με την απόφαση για τον τύπο της κλειδαριάς που απαιτεί ο χρήστης. Μετά, ακολουθούν διαφορετικοί αλγόριθμοι ανάλογα με τον τύπο αυτόν. Βεβαίως, το κόστος αυξάνεται αν αναλογιστούμε κι εκείνο που προκύπτει από τις άστοχες προβλέψεις στις οποίες μπορεί να καταλήξει ένας σημερινός επεξεργαστής που υλοποιεί *πρόβλεψη διακλάδωσης* (*branch prediction*) κατά την επιλογή του σωστού αλγορίθμου από την `pthreads`.

Κατ'επέκταση, η υλοποίηση της pthreads είναι αρκετά ασφαλής και δεν επιτρέπει εύκολα την κατάρρευση του συστήματος ή την κακή χρήση της μνήμης εκ μέρους της. Οπότε, ο επεξεργαστής αφιερώνει κι άλλους κύκλους που σχετίζονται με τις εντολές ελέγχου της ορθότητας των πράξεων που εκτέλεσε μια κλήση της pthreads, καθώς και της επιλογής κατάλληλου κωδικού σφάλματος αν αυτό υπάρξει.

Η παραπάνω λίστα μπορεί να αυξηθεί σε μέγεθος και για αυτό καταλήγουμε στην υπόθεση ότι ένας έμπειρος προγραμματιστής μπορεί να παρακάμψει τις βιβλιοθήκες και τις σχετικές καθυστερήσεις (*overhead*) και να καταφύγει στην ιδιόχειρη κατασκευή μεθόδων συγχρονισμού που στοχεύουν στις επιδόσεις και ταιριάζουν καλύτερα στις απαιτήσεις ανταγωνισμού μεταξύ νημάτων, μνήμης κ.ά. του εκάστοτε κώδικά του.

Τέλος, μπορεί να γίνει μια μικρή αναφορά στην έννοια της *συνέπειας μνήμης*

(*Memory Consistency*) και στον τρόπο που την εκμεταλλευόμαστε ώστε να βελτιστοποιήσουμε την απόδοση της εγγραφής/ανάγνωσης της μνήμης από τις διεργασίες ή τα νήματα.

3.6.1 Συνέπεια μνήμης

Ενώ τα πρωτόκολλα - πολιτικές συνοχής μνήμης, εγγυώνται την σωστή ενημέρωση της μνήμης κάθε επεξεργαστή ή ομάδας επεξεργαστών, τα πρωτόκολλα συνέπειας μνήμης ρυθμίζουν την χρονική στιγμή κατά την οποία οι τροποποιήσεις που πραγματοποιεί ένας επεξεργαστής σε μια κοινόχρηστη μεταβλητή θα γίνουν ορατές στους υπόλοιπους. Λέγοντας "ορατή" εννοούμε πως όλες οι μνήμες (κρυφές ή/και μη) έχουν λάβει τη νέα τιμή, αν έχουμε πρωτόκολλο εγγραφής-ενημέρωσης ή έχουν ακυρώσει το δεδομένο, αν έχουμε πρωτόκολλο εγγραφής-ακύρωσης για τη συνοχή. Φυσικά, τα πρωτόκολλα συνέπειας αφορούν εγγραφές κι αναγνώσεις που πραγματοποιούνται με ατομικές λειτουργίες και αφορούν συστήματα κοινόχρηστης μνήμης αλλά και κατανεμημένης.

Ας δούμε ένα παράδειγμα κώδικα. Υποθέτουμε πως οι αρχικές τιμές των A,B είναι 0:

```
1 ...
2 A=1;
3 B=1;
4 ...
```

Κώδικας 3.11: Κώδικας εγγραφής στη μνήμη

```
1 ...
2 printf("%d",B);
3 printf("%d",A);
4 ...
```

Κώδικας 3.12: Κώδικας ανάγνωσης της μνήμης

Αν υποθέσουμε πως οι κώδικες εκτελούνται παράλληλα από δύο νήματα, μπορούμε να φανταστούμε πως η εκτέλεση των εντολών μπορεί να συμβεί με μερικούς διαφορετικούς τρόπους. Η σειρά εκτέλεσης των εντολών μπορεί να οδηγήσει στην εμφάνιση των τιμών 10 ή ακόμα και 00 στην οθόνη. Αυτό όμως δε σημαίνει πως υπάρχει πρόβλημα με τον υπολογιστή. Το θέμα έγκειται στην ρύθμιση της συνέπειας της μνήμης.

Στην γενική περίπτωση, οι προγραμματιστές υποθέτουν πως το σύστημα θα υλοποιεί το πρωτόκολλο της *ακολουθιακής συνέπειας* (*Sequential Consistency*), σύμφωνα με το οποίο ένα νήμα A θα πρέπει να "περιμένει" ένα άλλο νήμα B να ολοκληρώσει

πλήρως την εκάστοτε τροποποίησή του σε μια κοινόχρηστη μεταβλητή μέσω ατομικών λειτουργιών, παρόλο που το νήμα Α δεν προορίζεται να την χρησιμοποιήσει. Λέγοντας ''ολοκληρώσει πλήρως'' εννόουμε πως όλο το σύστημα μνήμης θα πρέπει ''βλέπει'' την εγγραφή.

Το ακριβώς αντίθετο πρωτόκολλο είναι το πρωτόκολλο *χαλαρής συνέπειας* (*Relaxed Consistency*). Σύμφωνα με αυτό, οι ατομικές λειτουργίες χάνουν τον ''ατομικό'' χαρακτήρα τους και το σύστημα (μεταφραστής και υλικό) είναι ελεύθερο να πραγματοποιήσει βελτιστοποιήσεις στην σειρά των εσωτερικών εντολών ανάγνωσης-εγγραφής που πραγματοποιούνται σε μια κοινόχρηστη μεταβλητή μέσω ατομικών λειτουργιών. Στην καλή περίπτωση, μπορεί κάποιος επεξεργαστής να λάβει νωρίτερα μία ''πιο πρόσφατη'' εγγραφή από μία ''παλαιότερη'', καταστρατηγώντας έτσι την ακολουθιακή συνέπεια αλλά διατηρώντας παράλληλα τον ρυθμό διεκπεραίωσης του συστήματος υψηλό.

Στην προαναφερθείσα εκτέλεση των κωδίκων κι αν τηρηθεί ακολουθιακή συνέπεια, τότε η εκτέλεση της εντολής A=1; θα οδηγήσει στην εγγραφή αλλά και στην ενημέρωση της μνήμης. Οπότε, αυτό σημαίνει πως η εμφάνιση της τιμής 10 είναι αδύνατη καθώς προηγείται η ολοκληρωμένη διαδικασία εγγραφής της τιμής 1 στην μεταβλητή Α πριν την εγγραφή της μεταβλητής Β. Αν όμως τηρηθεί χαλαρή συνέπεια, τότε η εμφάνιση της τιμής 10 είναι πάλι πιθανή αφού επιτρέπεται η εκτέλεση εντολών εκτός της εμφανιζόμενης σειράς στον κώδικα.

Εκτός από τα δύο αυτά πρωτόκολλα, υπάρχουν μερικά ακόμη τα οποία κυμαίνονται σε αυστηρότητα μεταξύ των δύο προαναφερθέντων. Συγκεκριμένα, μέσω των `__atomic` κλήσεων του gcc έχουμε στη διάθεσή μας μια πληθώρα επιλογών εκτός από το το πρωτόκολλο της ακολουθιακής συνέπειας (επιλογή *__ATOMIC_SEQ_CST*) και χαλαρής συνέπειας (επιλογή *__ATOMIC_RELAXED*) [10]. Αυτές οι επιλογές είναι οι εξής:

- **__ATOMIC_ACQUIRE**: Δημιουργεί έναν περιορισμό «happens-before» μεταξύ του νήματος που εκτελεί μια ατομική λειτουργία ανάγνωσης που ακολουθεί αυτό το πρωτόκολλο και ενός άλλου νήματος που εκτελεί μια ατομική λειτουργία εγγραφής που εκτελεί το πρωτόκολλο συνέπειας *__ATOMIC_RELEASE*.

- **__ATOMIC_RELEASE**: Δημιουργεί έναν παρόμοιο περιορισμό με τον προηγούμενο, ώστε ένα νήμα που που εκτελεί μια ατομική λειτουργία ανάγνωσης που ακολουθεί το πρωτόκολλο *__ATOMIC_ACQUIRE* να δει την τροποποίηση

που πραγματοποίησε ένα άλλο νήμα νήμα μέσω ατομικής λειτουργίας εγγραφής που ακολουθεί το πρωτόκολλο συνέπειας *__ATOMIC_RELEASE*.

- **__ATOMIC_ACQ_REL**: Αποτελεί υβριδικό συνδυασμό μεταξύ των δύο προηγούμενων πολιτικών και μπορεί να θεωρηθεί σαν ”barrier” για την μνήμη και την ορατότητα των αλλαγών.

Αξιοσημείωτο είναι πως ενώ στην αρχιτεκτονική x86 μπορούν να χρησιμοποιηθούν πιο συγκεκριμένες πολιτικές (η πολιτική *__ATOMIC_HLE_RELEASE* και *__ATOMIC_HLE_RELEASE*), η αλήθεια είναι πως μπορεί να αγνοηθούν ολοτελώς αν η αρχιτεκτονική του συστήματος υποστηρίζει μόνο την ακολουθιακή συνέπεια. Για αυτό τα παραπάνω πρωτόκολλα δίνονται στο σύστημα περισσότερο σαν μορφή οδηγιών παρά εντολών.

Στην πράξη, αν θέλουμε να επιτρέψουμε στο σύστημα να εκτελέσει βελτιστοποιήσεις τότε μπορούμε να εφαρμόσουμε το πρωτόκολλο *__ATOMIC_RELAXED* σε κάθε ατομική λειτουργία. Αν όμως θέλουμε λίγο περισσότερο έλεγχο στην συνέπεια ή αν προκύπτουν σφάλματα, μπορούμε να κάνουμε συνδυασμούς χρήσης των πρωτοκόλλων *__ATOMIC_ACQUIRE* και *__ATOMIC_RELEASE*. Για την ακρίβεια, έχει νόημα να χρησιμοποιούμε την πολιτική *__ATOMIC_ACQUIRE* με την ατομική εντολή φόρτωσης από τη μνήμη *__atomic_load_n()*. Ταυτόχρονα, μπορούμε να κάνουμε χρήση της πολιτικής *__ATOMIC_RELEASE* με την επίσης ατομική εντολή αποθήκευσης στη μνήμη *__atomic_store_n()*. Αυτή τη σύμβαση ακολουθήσαμε στον προηγούμενο κώδικα.

Επίσης, στον προηγούμενο κώδικα εφαρμόσαμε την πολιτική *__ATOMIC_ACQ_REL* στην εντολή αφαίρεσης της `funlock()` καθώς και στην περίπτωση που επιτύχει η σύγκριση κι άρα η εναλλαγή στην εντολή *__atomic_compare_exchange_n()*. Στην περίπτωση που αποτύχει, επιλέγουμε το πρωτόκολλο χαλαρής συνέπειας, καθώς η περίπτωση αυτή δεν προκαλεί τροποποίηση της σημαντικής μεταβλητής της κλειδαριάς κι άρα επιτρέπουμε στο σύστημα να κάνει βελτιστοποιήσεις . Ωστόσο, η *__ATOMIC_ACQ_REL* προτιμάται όποτε η τροποποίηση συμβαίνει κι αφορά την τιμή της, η οποία πρέπει να γίνει γνωστή σε όποιο νήμα τη χρειαστεί άμεσα.

Κεφαλαιο 4

Ο OMPi και Το Συστημα Υποστηριξης Εκτελεσης OMPi Runtime

Το κεφάλαιο αυτό χωρίζεται σε τρία μέρη: μια επισκόπηση της διαδικασίας της μετασχηματισμού κώδικα του OMPi την ανάλυση της λειτουργίας του συστήματος υποστήριξης εκτέλεσης OMPi (*OMPi Runtime* ή `ORT`) και την μελέτη της ενσωμάτωσης των πειραματικών μεθόδων συγχρονισμού νημάτων στον κώδικά του. Υπενθυμίζουμε πως ο μεταφραστής OMPi μετατρέπει τις OpenMP οδηγίες σε αμιγώς παράλληλο C κώδικα, χρησιμοποιώντας κλήσεις της βιβλιοθήκης `ORT`. Υπενθυμίζουμε κιόλας πως η εργασία αυτή στοχεύει στην τροποποίηση της υπάρχουσας υλοποίησης ώστε οι οδηγίες συγχρονισμού OpenMP να αντικαθίστανται από κλήσεις που βασίζονται στις μεθόδους του 3ου Κεφαλαίου. Συγκεκριμένα, εστιάζουμε μόνο στον χρόνο εκτέλεσης, καθώς ο συγχρονισμός νημάτων σχετίζεται με τη φάση εκτέλεσης και όχι με τη φάση μετάφρασης.

4.1 Επισκόπηση της Διαδικασίας Μετάφρασης Κώδικα του OMPi

Ο OMPi μετασχηματίζει πηγαίο κώδικα σε C με οδηγίες OpenMP σε πολυνηματικό κώδικα C, βασισμένο αποκλειστικά σε μια από μερικές δοσμένες βιβλιοθήκες υποστήριξης εκτέλεσης ονόματι EELIB (*Execution Entity Library*). Η διαδικασία ξεκινά με συντακτική ανάλυση, η οποία περιλαμβάνει λεκτική ανάλυση και δημιουργία του συντακτικού δέντρου (*Abstract Syntax Tree* ή *AST*). Έπειτα, το δέντρο διασχίζεται για την αντικατάσταση των οδηγιών OpenMP με αντίστοιχες κλήσεις βιβλιοθηκών υποστήριξης. Το αποτέλεσμα είναι πηγαίος κώδικας σε C χωρίς οδηγίες OpenMP, αλλά με πολυνηματική λειτουργία υλοποιημένη εξ ολοκλήρου σε C. Στο τελικό στάδιο, ο παραγόμενος κώδικας μεταφράζεται μέσω του gcc και συνδέεται με τις βιβλιοθήκες υποστήριξης εκτέλεσης, ώστε να δημιουργηθεί το εκτελέσιμο αρχείο.

Στην πράξη, ο OMPi χρησιμοποιείται σε ένα σύστημα στο οποίο έχει εγκατασταθεί μέσω της εντολής ompicc στο bash. Για παράδειγμα: ompicc myprogram.c. Αν δεν ορισθεί αλλιώς, το παραγόμενο πρόγραμμα που δημιουργείται είναι το a.out και θα βρίσκεται στον ίδιο κατάλογο που κλήθηκε η εντολή. Χρησιμοποιώντας την εντολή ompicc για να μεταφράσουμε ένα πρόγραμμα ακολουθούνται κάποια βήματα:

1. Καλείται ο προεπιλεγμένος προεπεξεργαστής του συστήματος για τη γλώσσα C, που συνήθως είναι εκείνος που χρησιμοποιεί κι ο gcc (ο C Preprocessor ή cpp).

2. Καλείται ο επεξεργαστής/μετασχηματιστής _ompi, με τον οποίο παράγεται το αρχείο ενδιάμεσου κώδικα και περιέχει επίσης κώδικα C.

3. Καλείται ο προεπιλεγμένος μεταφραστής του συστήματος για τη γλώσσα C, που συνήθως είναι ο gcc, με τις κατάλληλες ρυθμίσεις (*options* ή *flags*) ενεργοποιημένες ώστε να μεταφραστεί το αρχείο ενδιάμεσου κώδικα του προηγούμενου βήματος (gcc -C) και να γίνει η σύνδεση (*linking*) με όσες βιβλιοθήκες του συστήματος είναι απαραίτητες αλλά και με την βιβλιοθήκη εκτέλεσης του OMPi (*OMPi Runtime Library*).

Αν θέλουμε κατά την διαδικασία μετάφρασης με τον OMPi να αποθηκευτεί το αρχείο ενδιάμεσου κώδικα που τελικά μεταφράζεται σε κώδικα μηχανής, μπορούμε να χρησιμοποιήσουμε την επιλογή -k. Για παράδειγμα: ompi -k myprogram.c. Αν

το πρόγραμμα που θέλουμε να μεταφραστεί έχει τον τίτλο `file.c` και χρησιμοποιήσουμε την εν λόγω επιλογή, τότε εκτός από το εκτελέσιμο αρχείο θα δημιουργηθεί και το αρχείο `file_ompi.c`, στο οποίο θα είναι αποθηκευμένος ο ζητούμενος ενδιάμεσος κώδικας. Στα περιεχόμενα του αρχείου αυτού υπάρχουν πάντα τα εξής:

- **Τα πρωτότυπα των συναρτήσεων** της βιβλιοθήκης εκτέλεσης, των οποίων τα ονόματα αρχίζουν με το πρόθεμα `ort_*` και υπάρχουν καθώς μπορεί να κληθούν αργότερα για διάφορους λόγους. Τα πρωτότυπα είναι πολλά σε πλήθος και για αυτόν τον λόγο το αρχείο ενδιάμεσου κώδικα είναι μεγάλο σε έκταση.

- **Ο κώδικας χρήστη**, που θυμίζει αρκετά τον αρχικό. Η αλήθεια είναι πως αν ο κώδικας δεν περιείχε καθόλου οδηγίες OpenMP, τότε είναι ακριβώς ο ίδιος. Στην αντίθετη περίπτωση, οι οδηγίες αντικαθίστανται από άλλες ισοδύναμες από τον `OMPi`, που συνολικά υλοποιούν τις αντίστοιχες λειτουργίες.

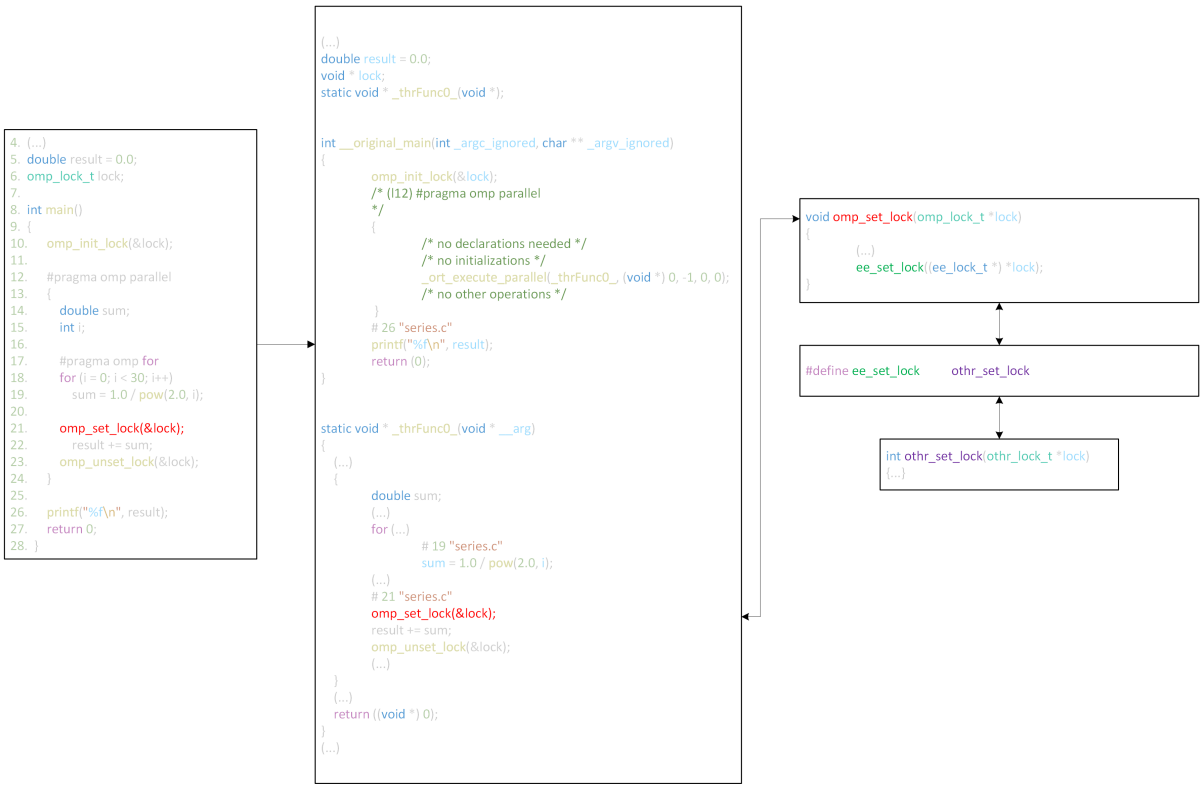
- **Η καινούργια main συνάρτηση του κώδικα χρήστη**, που ονομάζεται `__original_main()`. Εντός αυτής της συνάρτησης εκτελείται ο κώδικας χρήστη, ανεξάρτητα από το γεγονός αν υπήρχαν οδηγίες OpenMP και υπάρχουν αλλαγές.

- **Η main** συνάρτηση που δημιουργεί ο `OMPi`. Πρόκειται για απλή και μικρή συνάρτηση. Εκεί γίνεται η κλήση της `__original_main()` κι εκτελείται ο κώδικας χρήστη. Η κλήση αυτή περικλείεται από τις κλήσεις δύο άλλων συναρτήσεων:

  1. **ort_init()** για την σωστή αρχικοποίηση της βιβλιοθήκης χρόνου εκτέλεσης και διάφορων μεταβλητών.
  2. **ort_finalize()** για τον ασφαλή τερματισμό του συστήματος χρόνου εκτέλεσης.

Στο σχήμα 4.1 φαίνεται ένα παράδειγμα μετασχηματισμού της `main` ενός απλού προγράμματος (το αρχείο που την περιέχει ονομάζεται series.c) από τον `OMPi`, όπως φαίνεται στο αρχείο ενδιάμεσου κώδικα. Το πρόγραμμα αφορά τον υπολογισμό του αποτελέσματος του αθροίσματος των 30 πρώτων όρων της συγκλίνουσας άπειρης σειράς:

$$\sum_{i=0}^{n} \frac{1}{2^i}$$



Σχήμα 4.1: Μετασχηματισμός κώδικα από τον OMPi

Αποδεικνύεται πως η αριθμητική σειρά αυτή συγκλίνει στον αριθμό 2. Επιπλέον, όλες οι γραμμές σχολίων παρουσιάζονται αυτούσιες και δημιουργούνται από τον μεταφραστή.

Αρχικά, παρατηρούμε πως η συνάρτηση `main` έχει μετονομαστεί σε `__original_main()`, οι μεταβλητές δηλώνονται κι ορίζονται εκ νέου, η πράξη υπολογισμού κάθε όρου της σειράς μεταφέρθηκε σε άλλη συνάρτηση (`_thrFunc0_`), υπάρχουν καθοδηγητικά σχόλια που δείχνουν την αντικατάσταση των OpenMP οδηγιών και πως επειδή στο πρόγραμμα συναντάται παράλληλη περιοχή με την οδηγία `#pragma omp parallel`, καλείται η `ort_execute_parallel()` με κατάλληλα ορίσματα. Λεπτομέρειες για την συνάρτηση αυτή δίνονται στην επόμενη υποενότητα. Όσον αφορά το `struct __shvt__`, αρκεί να αναφέρουμε πως η πρόσβαση των κοινόχρηστων μεταβλητών της παράλληλης περιοχής (όχι οι global) από τα νήματα γίνεται με την χρήση του `struct` αυτού, το οποίο χρησιμοποιείται και σαν όρισμα στην `ort_execute_parallel()`.

Επιπλέον, στο σχήμα φαίνεται με απλοποιημένο τρόπο και η διαχείριση της συνάρτησης κλειδώματος του OpenMP `omp_set_lock()` από τον OMPi. Η αλήθεια είναι πως πράγματι η κλήση της συνάρτησης μεταφέρεται αυτούσια στο ενδιάμεσο αρχείο κώδικα. Ωστόσο, κατά την εκτέλεση του προγράμματος λαμβάνει χώρα η

αντικατάσταση της κλήσης αυτής και εκτελείται τελικά η αντίστοιχη συνάρτηση της βιλιοθήκης ORT. Στο εικονιζόμενο πρόγραμμα αντί της omp_set_lock(), βλέπουμε πως καλείται τελικά η othr_set_lock(). Αυτό συμβαίνει για κάθε συνάρτηση του OpenMP που χρησιμοποιεί ο χρήστης στο πρόγραμμά του.

Μπορούμε από τώρα να αναφέρουμε πως στην υλοποίησή μας, στον κώδικα της othr_set_lock() χρησιμοποιούνται futexes. Προτού όμως δωθούν όλες οι λεπτομέρειες της υλοποίησής μας, ακολουθεί μια σημαντική ανάλυση του συστήματος υποστήριξης χρόνου εκτέλεσης του OMPi.

4.2 Δομή και τρόπος λειτουργίας του OMPi Runtime

Το πλήρες σύστημα υποστήριξης χρόνου εκτέλεσης υλοποιείται στον κατάλογο runtime/ του πηγαίου κώδικα και περιλαμβάνει δύο κύρια υποσυστήματα:

- **Host subsystem**: Αντιπροσωπεύει το παραδοσιακό περιβάλλον χρόνου εκτέλεσης του OpenMP και λειτουργεί στους επεξεργαστές/πυρήνες του κύριου συστήματος (host). Εκεί εστιάζει η μελέτη της παρούσας εργασίας, καθώς εκεί πραγματοποιούνται οι προτεινόμενες τροποποιήσεις.
- **Devices subsystem**: Αποτελείται από εξειδικευμένες μονάδες οι οποίες:
  - Παρέχουν υποστήριξη χρόνου εκτέλεσης για κώδικες που εκτελούνται σε συνδεδεμένες συσκευές
  - Λειτουργούν ως διεπαφές μεταξύ του περιβάλλοντος χρόνου εκτέλεσης του host και των αντίστοιχων των συσκευών.

Ειδικότερα, το σύστημα υποστήριξης χρόνου εκτέλεσης του host του OMPi εντοπίζεται στον κατάλογο runtime/host/ και παρέχει και συντονίζει τις *οντότητες εκτέλεσης* (Execution Entities – EEs). Μια οντότητα χρόνου εκτέλεσης αποτελεί μια αφαίρεση για κάποια "οντότητα" (πχ. νήμα ή διεργασία), που τελικά εκτελεί τον εκάστοτε κώδικα αντί των νημάτων του OpenMP. Το σύστημα υποστήριξης χρόνου εκτέλεσης του host αποτελείται από δύο βασικές μονάδες:

1. Το ORT (runtime/host/ort.h και runtime/host/ort.c καθώς και runtime/host/parallel.c για τη διαχείριση παραλληλισμού), που διαχειρίζεται και χρονοδρομολογεί τις EEs και

2. Την εκάστοτε EELIB (`runtime/host/ee_*`), που είναι υπεύθυνη για τη δημιουργία και υλοποίηση των EEs. Υπάρχουν κάποιες EELIB βιβλιοθήκες διαθέσιμες, οι οποίες ακολουθούν κοινή διεπαφή, επιτρέποντας στο ORT να λειτουργεί ανεξάρτητα από την επιλεγμένη EELIB. Η επιλογή της EELIB γίνεται πριν την μετάφραση από τον χρήστη, αλλά δεν είναι υποχρεωτική για την εκτέλεση του προγράμματος.

Όταν πρόκειται να γίνει εκτέλεση μιας παράλληλης περιοχής, τότε καλείται η `ort_execute_parallel()`, μέσω της οποίας το ORT διαπραγματεύεται με την EELIB, μέσω της διεπαφής και συγκεκριμένων συναρτήσεων, για τον απαιτούμενο αριθμό EEs. Ο αριθμός και το είδος των EEs ποικίλλει ανάλογα με τις ρυθμίσεις του προγράμματος. Αφού εξασφαλιστούν, εκχωρούνται ως ομάδα (team).

Ταυτόχρονα, το ORT διατηρεί ένα δυναμικό δέντρο ελέγχου αποτελούμενο από *μπλοκ ελέγχου* (*EE Control Block* ή EECB) για την διαχείριση κάθε EE. Κάθε μπλοκ περιλαμβάνει στοιχεία όπως το επίπεδο παραλληλισμού, το μέγεθος μιας ομάδας, το αναγνωριστικό της και τον γονέα της. Δεν υπάρχει περιορισμός για το πλήθος των επιπέδων. Το δέντρο είναι δυναμικό καθώς επεκτείνεται με την δημιουργία καινούργιων ομάδων και συρρικνώνεται μόλις μια ομάδα ολοκληρώσει την εργασία του.

Κατά την αρχή της εκτέλεσης του προγράμματος, μόνο μία οντότητα εκτέλεσης (EE) είναι ενεργή και λειτουργεί στο επίπεδο 0. Αντιστοιχεί στην ρίζα του δέντρου των EECBs. Όταν μια EE συναντήσει παράλληλη περιοχή, γίνεται γονέας ή `master EE` της νέας ομάδας που δημιουργείται. Γενικά, αν μια EE βρίσκεται στο επίπεδο `i`, τα παιδιά της θα ανήκουν στο επίπεδο `i + 1`. Όπως αναφέρθηκε, δεν υπάρχει περιορισμός στο πλήθος των επιπέδων, επιτρέποντας πλήρη υποστήριξη εμφωλευμένου παραλληλισμού μόνο αν η EELIB μπορεί να παρέχει EEs. Αναφέρουμε πως η γονεϊκή οντότητα εκτέλεσης εντάσσεται στην ομάδα με αναγνωριστικό 0 και αποτελεί κανονικό μέλος της ομάδας των EEs όπως τα υπόλοιπα.

4.2.1 Η διεπαφή επικοινωνίας μεταξύ ORT και EELIBs

Η βιβλιοθήκη EELIB είναι υπεύθυνη για την παροχή όλων των οντοτήτων εκτέλεσης (EEs), εκτός της οντότητας - γονέα της ομάδας (master), καθώς και υποστήριξη για την υλοποίηση τριών τύπων κλειδώματος: `normal`, `nested` και `spin` (ο τελευταίος χρησιμοποιείται μόνο εσωτερικά από το ORT). Η EELIB δεν έχει άλλες υποχρεώσεις,

καθώς όλα τα υπόλοιπα είναι στην ευθύνη του ORT. Κατά την αρχικοποίηση, η EELIB ανακοινώνει στο ORT τις δυνατότητές της, όπως την υποστήριξη ένθετου (nested) παραλληλισμού, την δυναμική προσαρμογή πλήθους EEs, το μέγιστο πλήθος EEs και μέγιστο βάθος εμφωλευμένων επιπέδων παραλληλισμού.

Επιπλέον, η EELIB υλοποιεί τις συναρτήσεις `ee_request()`, `ee_create()` και `ee_waitall()`, τις οποίες καλεί το ORT. Αρχικά, η `ee_request()` χρησιμοποιείται από τον γονέα (master) για την αίτηση δημιουργίας ενός αριθμού νημάτων. Η συνάρτηση αυτή επιστρέφει τον αριθμό EEs που μπορούν τελικά να διατεθούν και ύστερα καλείται η `ee_create()` που δημιουργεί τελικά τις EEs μόνο αν η λειτουργικότητα υποστηρίζεται από την εκάστοτε EELIB (πχ. εμφωλευμένος παραλληλισμός). Επιπλέον, αν δεν υποστηρίζεται η δυναμική προσαρμογή του πλήθους των EEs και δεν διατείθενται αρκετές EEs, το πρόγραμμα τερματίζει αφού η `ee_request()` επιστρέφει 0.

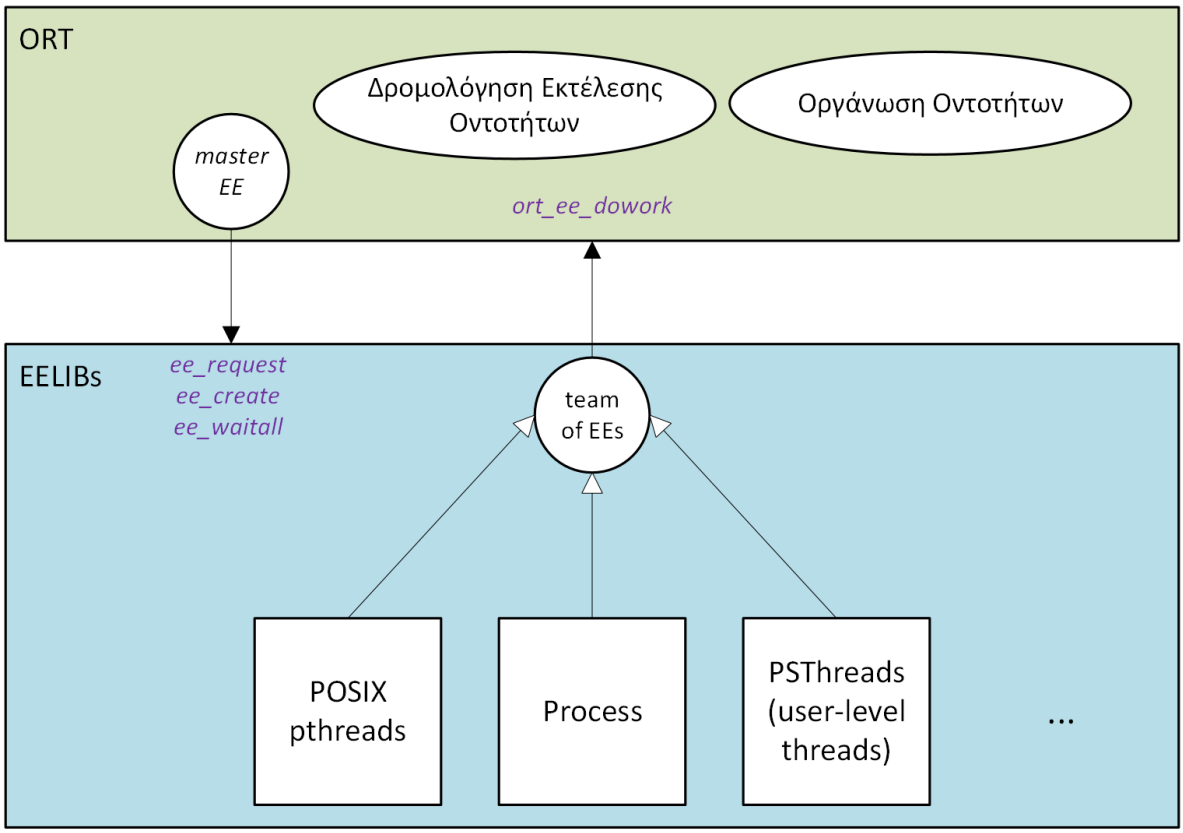
Τέλος, μετά την ολοκλήρωση της παράλληλης περιοχής, η `ee_waitall()` καλείται από τη οντότητα master και οπότε μπλοκάρει μέχρι να τερματίσουν όλα τα μέλη της ομάδας την εργασία τους.

4.2.2 EELIBs

Η έκδοση 3.0 του OMPi συνοδεύεται από μια EELIB νημάτων POSIX, μια EELIB νημάτων επιπέδου χρήστη (*user-level threads* κι ονομάζεται `psthreads`) κι άλλη μια EELIB που αφορά διεργασίες. Ο αριθμός των διαθέσιμων EELIB μπορεί να επεκταθεί καθώς στον πηγαίο κώδικα (`runtime/host/LibraryTemplate`) δίνονται τα απαραίτητα αρχεία κι οδηγίες επέκτασης για τον χρήστη.

Το σχήμα 4.2 περιγράφει συνοπτικά την σχέση μεταξύ των δύο συστατικών του συστήματος υποστήριξης χρόνου εκτέλεσης του `host`. Σε αυτό το σχήμα φαίνονται και οι διαθέσιμες EELIBS της έκδοσης 3.0 του OMPi.

Η προεπιλεγμένη (default) βιβλιοθήκη του OMPi χρησιμοποιεί μια δεξαμενή (`pool`) από `POSIX threads` ως EEs. Η δεξαμενή δημιουργείται κατά την αρχικοποίηση, και το πλήθος των νημάτων είναι `N`, δηλαδή το μέγιστο μεταξύ των διαθέσιμων επεξεργαστών και της τιμής της μεταβλητής περιβάλλοντος `OMP_NUM_THREADS`. Μετά τη δημιουργία της, το μέγεθος της δεξαμενής δεν αλλάζει. Τα νήματα παραμένουν ενεργά, αποδίδοντας συχνά τον επεξεργαστή, αναμένοντας εργασία. Κατά την αίτηση EEs, ανεξάρτητα από το επίπεδο παραλληλισμού, η EELIB ελέγχει τη διαθεσιμότητα στη



Σχήμα 4.2: Το σύστημα υποστήριξης χρόνου εκτέλεσης του host

δεξαμενή· μπορεί να δώσει μόνο όσα νήματα είναι άμεσα διαθέσιμα. Έτσι, ο εμφωλευμένος παραλληλισμός υποστηρίζεται μόνο όταν το σύνολο των νημάτων όλων των επιπέδων δεν υπερβαίνει το `N`.

Αν έχει απενεργοποιηθεί η δυναμική προσαρμογή από τον προγραμματιστή, η βιβλιοθήκη ενδέχεται να μην μπορεί να καλύψει την απαίτηση πλήθους νημάτων. Μετά την ολοκλήρωση της εργασίας του, κάθε νήμα επιστρέφει στη δεξαμενή, μειώνοντας έναν σχετικό μετρητή στο `EECB` του γονέα. Το νήμα-γονέας (`master`) μπλοκάρει στη `ee_waitall()` μέχρι ο μετρητής να γίνει 0. Τότε είναι που όλα τα νήματα της ομάδας θα έχουν ολοκληρώσει την εργασία τους.

4.3 Λεπτομέρειες σχετικές με τις EELIBs

Αντί να παραθέσουμε κατευθείαν την τοποθεσία των τροποποιήσεων στους καταλόγους του πηγαίου κώδικα και το περιεχόμενό τους, προηγείται μια βασική ανάλυσηΣ των απαιτήσεων που ορίζει το `ORT` σε μια οποιαδήποτε `EELIB` νημάτων

ώστε να είναι σε θέση να την χρησιμοποιήσει, καθώς και του τρόπου που ικανοποιεί τις απαιτήσεις αυτές.

Συγκεκριμένα, οι απαιτήσεις συνοψίζονται στην παρακάτω λίστα:

1. **Τις συναρτήσεις διαχείρισης κλειδαριών συγχρονισμού των EEs και την υλοποίησή τους** και
2. **Τις συναρτήσεις αναμονής ολοκλήρωσης (`joins`) και δημιουργίας των EEs και την υλοποίησή τους**

Προαιρετικά, η `EELIB` μπορεί να παρέχει και συναρτήσεις - υλοποίηση για `barriers` και `tasks`.

Επίσης, ο κώδικας που υλοποιεί την κάθε `EELIB` πρέπει να εμπεριέχεται σε δύο αρχεία κώδικα C, με τους εικονιζόμενους τίτλους:

1. **ee.h**, δηλαδή το αρχείο κεφαλίδας (`header file`) που περιέχει τα πρωτότυπα συναρτήσεων, τα `macros`, τις αρχικοποιήσεις μεταβλητών κ.ά.
2. **othr.c**, το αρχείο στο οποίο υπάρχει ο κώδικας των συναρτήσεων που αναφέρονται στο `ee.h`.

Τα δύο παραπάνω αρχεία υπάρχουν σε κάθε κατάλογο που αντιστοιχεί σε μια `EELIB`. Οι κατάλογοι αυτοί βρίσκονται εντός του καταλόγου `runtime/host/libort` και η ονομασία τους έχει την μορφή `ee_*` (πχ. `ee_pthreads/`).

Όσον αφορά τις συναρτήσεις συγχρονισμού, κάθε βιβλιοθήκη πρέπει να παρέχει τις εξής πέντε συναρτήσεις:

1. `int othr_init_lock(othr_lock_t *lock, int kind);`
   Αρχικοποιεί μια κλειδαριά - αντικείμενο τύπου `othr_lock_t`, αναλόγως με τον επιλεγμένο τύπο κλειδαριάς (`normal, spin, nested`).
2. `int othr_destroy_lock(othr_lock_t *lock);`
   Καταστρέφει μια κλειδαριά που έχει προηγουμένως αρχικοποιηθεί αποδεσμεύοντας την μνήμη που κατελάμβανε και κάνοντας οποιεσδήποτε άλλες σχετικές ενέργειες.
3. `int othr_set_lock(othr_lock_t *lock);`
   Πραγματοποιεί το κλείδωμα. Αν η κλειδαριά είναι ήδη κατειλημμένη/κλειδωμένη, προκαλεί το μπλοκάρισμα του καλούντος νήματος μέχρι να γίνει πάλι διαθέσιμη.

4. `int othr_unset_lock(othr_lock_t *lock);`
   Απελευθερώνει μια κλειδαριά που προηγουμένως είχε κλειδωθεί και προκαλεί το ξεμπλοκάρισμα ενός μπλοκαρισμένου νήματος.

5. `int othr_test_lock(othr_lock_t *lock);`
   Ελέγχει αν η κλειδαριά είναι διαθέσιμη και αν ναι τότε την καταλαμβάνει χωρίς να προκαλεί μπλοκάρισμα.

Όπως φαίνεται αργότερα στο παρόν κεφάλαιο, η ουσία των τροποποιήσεων που προτείνονται στην εργασία αυτή συμβαίνει στον κώδικα των παραπάνω πέντε συναρτήσεων. Σημαντικό είναι να παρατηρήσουμε πως δεν προσφέρονται συναρτήσεις μεταβλητών συνθήκης. Αυτές, χρησιμοποιούνται εσωτερικά στις παραπάνω συναρτήσεις.

Οι τύποι κλειδαριών `normal` και `nested` είναι διαθέσιμοι στους χρήστες μέσω των προγραμμάτων OpenMP, ενώ τα `spin locks` χρησιμοποιούνται εσωτερικά από το `ORT`. Το είδος του κλειδώματος καθορίζεται κατά την αρχικοποίησή του μέσω της `othr_init_lock()` , όπου η δεύτερη παράμετρός της μπορεί να είναι μία από τις τιμές `ORT_LOCK_NORMAL`, `ORT_LOCK_NEST`, ή `ORT_LOCK_SPIN`. Ωστόσο, η βιβλιοθήκη πρέπει να παρέχει έναν ενιαίο τύπο για κάθε τύπο κλειδαριάς: τον `othr_lock_t`. Επίσης, πρέπει να ορίζεται και η δομή `othr_nestlock_t` για την διαχείριση των `nested` κλειδαριών. Εν τέλει όμως, χρησιμοποιείται μόνο ο `othr_lock_t`, στον οποίο υπάρχει διαθέσιμος χώρος για την αποθήκευση δεδομένων μόνο ενός τύπου κλειδαριάς (χρήση `union`). Υπάρχει σχετικό παράδειγμα κώδικα στον Κώδικα 4.1, στο οποίο περιγράφεται ένα αρχείο `ee.h` και περιλαμβάνει ό,τι έχει συζητηθεί στην παράγραφο αυτή.

```
(...Macros...)
typedef struct                  /* For nested locks */
{
  pthread_mutex_t lock;      /* The lock in question */
  pthread_mutex_t ilock;     /* Lock to access the whole struct */
  pthread_cond_t  cond;      /* For waiting until lock is available */
  int             count;     /* # times locked by the same thread */
  void           *owner;     /* The owner (task) of the nestable lock
     */
} othr_nestlock_t;

typedef union
{
  struct
  {
    int   type;                 /* normal/spin/nested */
    union
    {
      pthread_mutex_t normal;   /* normal lock */
      othr_nestlock_t nest;     /* nest lock */
#ifdef HAVE_SPINLOCKS           /* spin lock */
      pthread_spinlock_t spin;
#else
      struct
      {
        int             rndelay;     /* Used for initial spin delays
           */
        pthread_mutex_t mutex;
      } spin;
#endif
    } data;
  } lock;
} othr_lock_t;
(... Functions ...)
```

Κώδικας 4.1: Μέρος του αρχείου κώδικα ee.h της ee_pthreads

Τέλος, όσον αφορά τις συναρτήσεις αναμονής ολοκλήρωσης (joins) και δημιουργίας των EEs, εστιάζουμε στην περίπτωση χρήσης νημάτων στην οποία το ORT απαιτεί μόνο τρεις συναρτήσεις για τη διαχείριση των οντοτήτων εκτέλεσης (EEs):

- `int othr_request(int numees, int level)`, όπου `numees` πρέπει να αποτελεί το πλήθος των ζητούμενων οντοτήτων και `level` το επίπεδο εμφωλευμένης παραλληλίας της αρχικής οντότητας. Η συνάρτηση επιστρέφει τον αριθμό των οντοτήτων που τελικά η EELIB μπορεί να προσφέρει.

- `void othr_create(int numees, int level, void *arg, void **info)`, όπου το όρισμα `level` ακολουθεί την ίδια λογική με την προηγούμενη συνάρτηση και το `numees` πρέπει να αντιστοιχεί στον αριθμό των οντοτήτων που η EELIB μπορεί πράγματι να προσφέρει (βλ. `othr_request()`). Τα υπόλοιπα δύο ορίσματα σχετίζονται με την μεταφορά πληροφοριών της εργασίας που πρέπει να εκτελεστεί (πχ. ποιες είναι οι κοινόχρηστες κλειδαριές) και με την ταυτότητα της ομάδας των EEs.

- `void othr_waitall(void **info)`, η οποία απαιτεί μόνο ένα όρισμα που συνδέεται με τις πληροφορίες του γονέα-EE, όπως την ταυτότητά του ή το πλήθος των παιδιών του. Με αυτήν την συνάρτηση προκαλείται η αναμονή του γονέα μέχρι τα παιδιά του να ολοκληρώσουν τις εργασίες τους.

Οι παραπάνω τρεις συναρτήσεις αντιστοιχούν πλήρως στις `ee_request()`, `ee_create()` και `ee_waitall()` που έχουν αναφερθεί. Σαφώς η αιτία που δεν καλούνται οι παραπάνω `othr_*` συναρτήσεις αντί για τις `ee_*` είναι ο βασικός στόχος της ευελιξίας και κλιμακωσιμότητας που προσφέρει ο OMPi. Για παράδειγμα, ενώ το ORT μπορεί πάντα να χρησιμοποιεί τις `ee_*` συναρτήσεις, το σύστημα μπορεί να ρυθμιστεί έτσι ώστε οι EEs να μην είναι νήματα αλλά διεργασίες. Τότε, πρέπει να τελικά να χρησιμοποιηθούν συναρτήσεις από την αντίστοιχη EELIB η οποία θα περιέχει συναρτήσεις της μορφής `oprc_*` αλλά χωρίς το ORT να το "γνωρίζει". Έτσι ο μεταφραστής παραμένει κλιμακώσιμος. Το ίδιο σκεπτικό ακολουθείται για κάθε συνάρτηση μιας EELIB.

Σε αυτό το σημείο, έχουμε παρουσιάσει τις πιο βασικές και ταυτόχρονα πιο χρήσιμες για την μελέτη μας συναρτήσεις και χαρακτηριστικά που αφορούν τις EELIBs. Μόνο τώρα μπορεί να ακολουθήσει η υποενότητα στην οποία γίνεται η ανάλυση των τροποποίησεων που έλαβαν χώρα στα αρχεία μιας EELIB και είχαν σαν αποτέλεσμα την ενσωμάτωση του μηχανισμού των futexes.

4.4 Ενσωμάτωση του μηχανισμού των futexes στο runtime σύστημα του OMPi

Σε αυτή την ενότητα εξετάζουμε την ενσωμάτωση των πειραματικών μεθόδων συγχρονισμού νημάτων του Κεφαλαίου 3 που χρησιμοποιούν `futexes` και στοχεύουν στην μείωση κλήσεων του πυρήνα. Ο τελικός κώδικας των τροποιημένων συναρτήσεων του αρχείου `othr.c` και των τροποποιήσεων του αρχείου `ee.h` βρίσκεται στο παράρτημα κώδικα. Στην παρούσα υποενότητα δίνεται η περιγραφή των αλλαγών που πραγματοποιήθηκαν.

Αρχικά, οι αλλαγές έγιναν στα αρχεία της ήδη υπάρχουσας και λειτουργικής `EELIB` που σχετίζεται με pthreads: την `ee_pthreads`. Δεν χρειάστηκε η δημιουργία μιας καινούργιας `EELIB`. Το πρώτο βήμα ήταν η προσθήκη των πρωτότυπων των μεθόδων που υποστηρίζουν και χρησιμοποιούν `futexes` (κώδικας 3.2, 3.5, 3.6, 3.7, 3.8, 3.9) στο αρχείο `ee.h`. Ύστερα τοποθετήθηκε και ο κώδικάς τους στο `othr.c`.

Σημαντικό γεγονός είναι πως δεν τροποποιήθηκε καμιά από τις υπογραφές των συναρτήσεων συγχρονισμού νημάτων και ισχύουν αυτές που περιγράφτηκαν στην προηγούμενη υποενότητα. Το ίδιο ισχύει και για τις τιμές που επιστρέφουν. Αρκούσε η αντικατάσταση των κλήσεων `pthread_mutex_*` και `pthread_cond_*` εντός του κώδικα αυτών, που προϋπάρχουν στο `othr.c`, με τις αντίστοιχες που περιγράφονται στο Κεφάλαιο 3. Για παράδειγμα, μια κλήση της `pthread_mutex_lock(mutex)` αντικαθίσταται από την `flock(futexp)`.

Ωστόσο, για την υποστήριξη της λειτουργικότητας των εν λόγω αντικαταστάσεων χρειάστηκε η τροποποίηση των δομών `othr_nestlock_t` και `othr_lock_t` που περιγράφουν τις κλειδαριές στο αρχείο `ee.h`. Ακολουθώντας το πνεύμα των προηγούμενων αντικαταστάσεων, κάθε τύπος `pthread_mutex_t` και `pthread_cond_t` αντικαταστάθηκε από τον τύπο `uint32_t`, που αρκεί για την υποστήριξη των `futex calls` που έχουμε εντάξει στον κώδικα.

Επιπλέον, όσον αφορά τα `spinlocks` διατηρούμε μόνο τον μετρητή καθυστέρησης αναμονής(`int rndelay`) και το `mutex`. Η μοναδική διαφορά είναι πως ο τύπος του `mutex` είναι πλέον `uint32_t` κι όχι `pthread_mutex_t`. Αυτή η λύση παραμένει αποτελεσματική, εύκολη στην κατανόηση και συμβατή με τη χρήση `futexes`.

Ακολουθεί μια περιγραφή των τροποποιήσεων κι αντικαταστάσεων ανά συνάρτηση συγχρονισμού νημάτων του `othr.c`. Σημειώνουμε πως κάθε συνάρτηση πρέπει να επιστρέφει 0 στην περίπτωση επιτυχίας και πως τα ορίσματα παραλείπονται

καθώς δεν έχει αλλαχτεί ούτε ο τύπος τους ούτε το πλήθος τους.

- `int othr_init_lock()` : Κάθε μεταβλητή που σχετίζεται με τον συγχρονισμό νημάτων αποτελεί ακέραιο αριθμό, ο οποίος αρχικοποιείται στο 0. Το μόνο που διαφέρει ανά περίπτωση τύπου κλειδαριάς είναι το πλήθος των αρχικοποιήσεων. Για παράδειγμα, ενώ στην περίπτωση της απλής κλειδαριάς αρκεί η αρχικοποίηση μιας ακέραιας μεταβλητής, στην περίπτωση της εμφωλευμένης κλειδαριάς για εμφωλευμένο παραλληλισμό πρέπει να αρχικοποιηθούν τέσσερις ακέραιες μεταβλητές.

- `int othr_destroy_lock()` : Ακολουθώντας την ίδια λογική με την περιγραφή της προηγούμενης συνάρτησης, αρκεί η επιστροφή της τιμής 0 καθώς δεν υπάρχει περιοχή μνήμης της οποίας η δέσμευση έγινε δυναμικά και πρέπει να αποδεσμευτεί.

- `int othr_set_lock()` : Χωρίς να παραβιάζεται ο τρόπος λειτουργίας του τρόπου απόκτησης της κλειδαριάς, κάθε κλήση συγχρονισμού της `pthreads` έχει αντικατασταθεί από μια ισοδύναμη πειραματική μέθοδο του Κεφαλαίου 3. Εδώ, για την υλοποίηση των `spinlocks` έχει διατηρηθεί και τροποποιηθεί μόνο η γενική φορητή μέθοδος αναμονής με εκθετική οπισθοχώρηση.

- `int othr_unset_lock()` : Πραγματοποιήθηκαν παρόμοιες αντικαταστάσεις με την προηγούμενη περίπτωση.

- `int othr_test_lock()` : Πραγματοποιήθηκαν παρόμοιες αντικαταστάσεις για άλλη μια φορά.

Σαφώς ο τρόπος λειτουργίας των παραπάνω πέντε συναρτήσεων πρέπει να έχει γίνει κατανοητός πριν την εφαρμογή των προτεινόμενων τροποποιήσεων. Χωρίς να αναλωθούμε σε λεπτομέρειες, αρκεί να παρατηρήσουμε πως σε κάθε συνάρτηση (εκτός από την `othr_destroy_lock()`) υπάρχει μια δομή `switch` η οποία εξετάζει την τιμή της μεταβλητής `type` που συνοδεύει την δομή `othr_lock_t` την οποία λαμβάνει ως όρισμα. Σε σειρά φθίνουσας πολυπλοκότητας, θα μπορούσαμε να κατατάξουμε πρώτη την περίπτωση των εμφωλευμένων κλειδαριών, όπου στις συναρτήσεις κλειδώματος και ξεκλειδώματος υπάρχουν κρίσιμες περιοχές, στις οποίες η πρόσβαση γίνεται μέσω άλλης κλειδαριάς (`ilock`). Οι κρίσιμες περιοχές αυτές είναι απαραίτητες για τον υπολογισμό των τιμών του μετρητή που σχετίζεται με αυτές (`count`)

και τον ορισμό του καινούργιου νήματος-ιδιοκτήτη. Επιπλέον, σε αυτόν τον τύπο κλειδώματος χρησιμοποιούνται και οι μεταβλητές συνθήκης, για τις περιπτώσεις που ο ιδιοκτήτης της κλειδαριάς δεν είναι το νήμα που προσπαθεί να αποκτήσει τον έλεγχό της και τελικά πρέπει να μπλοκαριστεί.

Ο επόμενος τύπος κλειδαριάς στην κατάταξη είναι τα spinlocks, που αρκεί να σημειώσουμε πως σε κάθε αποτυχημένη προσπάθεια κλειδώματος (μέσω της int othr_set_lock()) από ένα νήμα το αναγκάζει να αναμένει για κάποιο χρονικό διάστημα σπαταλώντας κύκλους της CPU. Τέλος ακολουθούν τα απλά και πολυσυζητημένα mutex.

Μια τελευταία παρατήρηση αφορά τη διαφορά μεταξύ των nested/εμφωλευμένων locks που υποστηρίζει ο OMPi και των recursive locks που συναντήσαμε στο Κεφάλαιο 3. Στην πράξη, ο OMPi υποστηρίζει τη λειτουργικότητα των recursive locks μέσω του τύπου ORT_LOCK_NEST, στην οποία περίπτωση ακολυθείται το πρότυπο POSIX και το κλείδωμα πετυχαίνει όταν η κλειδαριά είναι ελεύθερη και μπορεί να επαναληφθεί από το ίδιο νήμα. Σαφώς, ένα νήμα μπορεί να είναι ιδιοκτήτης πολλών κλειδαριών, ανεξαρτήτως τύπου. Ωστόσο, για το ξεκλείδωμα απαιτείται το ίδιο πλήθος κλήσεων ξεκλειδώματος με το πλήθος κλήσεων κλειδώματος που πραγματοποιήθηκαν.

Κεφαλαιο 5

Εκτελεση πειραματων και Παρουσιαση Αποτελεσματων

Σε αυτό το κεφάλαιο γίνεται ανάλυση του περιβάλλοντος εκτέλεσης των πειραμάτων, του τρόπου που το περιβάλλον αυτό χρησιμοποιήθηκε για την εκτέλεση των πειραμάτων αξιολόγησης και η παρουσίαση των αποτελεσμάτων τους. Τα πειράματα εκτέλεσθηκαν σε ένα από τα συστήματα που είναι στη διάθεση του Parallel Processing Group (PPG) του Τμήματος Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής του Πανεπιστημίου Ιωαννίνων και αφορούσαν την εκτίμηση διάφορων μετρικών της μεθοδικής εκτέλεσης μιας σουίτας μετροπρογραμμάτων (*benchmarks*) και μιας σουίτας εφαρμογών στο εν λόγω σύστημα. Ακολουθεί αρχικά μια περιγραφή του υπολογιστικού συστήματος που χρησιμοποιήθηκε για τα πειράματα και μια λεπτομερής ανάλυση της ταυτότητας και της μεθοδολογίας εκτέλεσης αυτών

και κατόπιν λεπτομέρειες της εκτέλεσης των πειραμάτων. Τέλος, ακολουθεί παρουσίαση των αποτελεσμάτων και το κεφάλαιο καταλήγει με τον σχολιασμό τους.

5.1 Τεχνικά Χαρακτηριστικά του Περιβάλλοντος Εκτέλεσης

Για την εκτέλεση των πειραμάτων και την εκτίμηση της συνεισφοράς των `futexes` στην βελτίωση της επίδοσης των μεταφρασμένων προγραμμάτων που προκύπτουν από τον τροποποιημένο OMPi, επιστρατεύθηκε ένας ισχυρός υπολογιστής που ιδανικά προορίζεται για τον ρόλο του εξυπηρετητή ενός δικτύου υπολογιστών (*Rack Mounted Server*) ή την ανάλυση μεγάλου όγκου δεδομένων. Πρόκειται για τον *Dell EMC PowerEdge R840* και το λειτουργικό σύστημα ανήκει στην οικογένεια των Linux.

Καθώς η φύση των πειραμάτων αφορά την μέτρηση καθυστερήσεων, είναι απαραίτητο να γνωρίζουμε τις καθυστερήσεις επικοινωνίας μεταξύ επεξεργαστών και μνήμης. Για αυτό, πρέπει να γνωρίζουμε την αρχιτεκτονική που εφαρμόζεται στο σύστημα και ο ακριβής τρόπος να την ανακαλύψουμε είναι μέσω της βιβλιοθήκης *hwloc* (*Hardware Locality*). Ωστόσο, χρήσιμο είναι και το εργαλείο `lscpu`, που είναι εγκατεστημένο στα περισσότερα Linux λειτουργικά συστήματα και ακόμα κι αν χρησιμοποιηθεί χωρίς επιπλέον `flags`, επιστρέφει μια συνολική εικόνα του συστήματος. Άλλο εργαλείο τερματικού αποτελεί η εντολή `numactl` και τα `flags` της, που χρησιμοποιείται για διάφορες λειτουργίες σχετικές με διαχείριση NUMA. Ο πίνακας 5.1 συνοψίζει τα πιο σημαντικά χαρακτηριστικά του συστήματος.

Με απλά λόγια, το σύστημα διαθέτει τέσσερις επεξεργαστές `Intel Xeon Gold 6130` και κάθε ένας από αυτούς διαθέτει 16 πυρήνες. Κάθε πυρήνας διαθέτει κρυφή μνήμη εντολών, κρυφή μνήμη δεδομένων, ένα δεύτερο επίπεδο κρυφής μνήμης και συνολικά όλοι οι πυρήνες μοιράζονται κι ένα ακόμα επίπεδο κρυφής μνήμης μεγάλης χωρητικότητας. Λόγω της ενεργοποιημένης δυνατότητας `HyperThreading`, κάθε πυρήνας υποστηρίζει την εκτέλεση έως 2 νημάτων (2 Threads per core σύμφωνα και με τον πίνακα 5.1). Αυτό σημαίνει πως κάθε επεξεργαστής διαθέτει αρκετούς καταχωρητές για την αποθήκευση δεδομένων δύο νημάτων καθώς στόχος της λειτουργίας αυτής είναι η εισαγωγή εντολών στη διοχέτευση ενός πυρήνα που είναι ανεξάρτητες μεταξύ τους κι άρα μπορούν να εκτελεστούν παράλληλα σε μια υπερβαθμωτή διοχέτευση. Οπότε, το σύστημα υποστηρίζει την ταυτόχρονη εκτέλεση 128

Πίνακας 5.1: Τεχνικά Χαρακτηριστικά Συστήματος

Threads	128
Threads per core	2
Cores per socket	16
Model name	Intel Xeon Gold 6130 CPU
Base Clock Rate	2.10GHz
Sockets	4
L1d cache	32 KiB (ανά πυρήνα)
L1i cache	32 KiB (ανά πυρήνα)
L2 cache	1024 KiB (ανά πυρήνα)
L3 cache	22 MiB (ανά socket)
NUMA nodes	4

Πίνακας 5.2: Καθυστερήσεις επικοινωνίας μεταξύ των NUMA Κόμβων (συμβολική μονάδα μέτρησης)

NUMA Node	**0**	**1**	**2**	**3**
0	10	21	21	21
1	21	10	21	21
2	21	21	10	21
3	21	21	21	10

νημάτων.

Όσον αφορά την επικοινωνία μεταξύ επεξεργαστών και της μνήμης, η αρχιτεκτονική που εφαρμόζεται είναι η `NUMA`. Επιπλέον, με την βοήθεια της `hwloc` και της εντολής τερματικού bash/προγράμματος `lstopo` που αυτή προσφέρει, βλέπουμε πως κάθε `socket` αποτελεί ένα ξεχωριστό `NUMA Node`, με ιδιωτική μνήμη 63 GiB (εκτός από το Node 0 που διαθέτει 62 GiB). Εξ ορισμού λοιπόν οι επεξεργαστές - Nodes αντιλαμβάνονται πως υπάρχει μια μοναδική κοινόχρηστη μνήμη και η μόνη "απόδειξη" που θα μπορούσαν να έχουν ότι εφαρμόζεται `NUMA` αρχιτεκτονική θα αποτελούσε η καθυστέρηση που μπορεί να μεσολαβήσει για την ανάγνωση κάποιων δεδομένων· όσων δεν βρίσκονται στην ιδιωτική μνήμη του κόμβου. Άλλωστε, εκτελώντας `numactl -H` στο τερματικό μπορούμε να δούμε το (συμβολικό) κόστος επικοινωνίας μεταξύ των `nodes`. Οι σχετικές πληροφορίες που αντλούμε μέσω της εντολής αυτής

Πίνακας 5.3: Αρίθμηση λογικών πυρήνων των NUMA Κόμβων

NUMA Node	**Logical Cores (physical index)**
0	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
1	1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61
2	2 6 10 14 18 22 26 30 34 38 42 46 50 54 58 62
3	3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63

παρατείθενται στον πίνακα 5.2. Σύμφωνα με τον πίνακα, η καθυστέρηση επικοινωνίας μεταξύ των επεξεργαστών και της ιδιωτικής μνήμης ενός κόμβου θα πρέπει να ισούται περίπου με την μισή από αυτήν που παρατηρείται στην επικοινωνία μεταξύ ενός κόμβου και της μνήμης ενός οποιουδήποτε άλλου κόμβου εκτός του ίδιου.

Επιπλέον, ανακαλύπτουμε πως κάθε κόμβος περιέχει ένα πλήθος από "λογικούς πυρήνες" (`logical cores`), των οποίων την έννοια την συναντήσαμε περιληπτικά στο πρώτο Κεφάλαιο. Το σύστημα τους αντιλαμβάνεται ως φυσικούς/κατασκευασμένους επεξεργαστές-πυρήνες, παρόλο που οι τελευταίοι είναι λιγότεροι (οι μισοί συνολικά σε πλήθος). Στο εξής, ο όρος "πυρήνας" θα αναφέρεται σε μια από τις δεκαέξι κατασκευασμένες υπαρκτές επεξεργαστικές μονάδες που διαθέτει κάθε `socket`. Στην περίπτωση που θέλουμε να αναφερθούμε στον "*πυρήνα*" που βλέπει το σύστημα λόγω της δυνατότητας HyperThreading, θα χρησιμοποιούμε τον όρο "λογικός" ή "*εικονικός πυρήνας*". Εύκολα καταλαβαίνουμε πως σε κάθε πυρήνα αντιστοιχούν δύο εικονικοί πυρήνες στο συγκεκριμένο σύστημα.

Κατ'επέκταση, ανακαλύπτουμε πως σε κάθε εικονικό πυρήνα έχουν ανατεθεί δύο δείκτες: έναν που αναθέτει το λειτουργικό σύστημα (`physical/OS index`) κι έναν που αναθέτει η `hwloc` (`logical index`). Αυτό συμβαίνει επειδή ένας από τους σκοπούς της *hwloc* είναι η διευκόλυνση διαδικασίας της *αντίστροφης αναζήτησης* (*reverse search*) ενός πυρήνα σε όλο το σύστημα. Ωστόσο, ασφαλώς επειδή το λειτουργικό σύστημα χρησιμοποιεί τους δείκτες που αναθέτει το ίδιο, γίνεται χρήση μόνο των δεικτών που αναθέτει το λειτουργικό σύστημα στους πυρήνες στα πλαίσια της εργασίας αυτής.

Οι δείκτες των εικονικών πυρήνων που περιέχονται σε κάθε `NUMA Node` βρίσκονται στον συνοπτικό πίνακα 5.3 και οι δείκτες που εμφανίζονται είναι εκείνοι που αναθέτει και χρησιμοποιεί το λειτουργικό σύστημα. Αυτός ο πίνακας είναι πολύ χρήσιμο σημείο αναφοράς για την περιγραφή των πειραμάτων που ακολουθεί.

Για μια περεταίρω επιβεβαίωση της εικόνας της αρχιτεκτονικής που μας δίνουν τα εργαλεία και οι εντολές, στο παράρτημα κώδικα δίνεται ένα `bash script`, το οποίο εκτελεί με πολυνηματική προσέγγιση ένα μετροπρόγραμμα [11] σχεδιασμένο για την εκτίμηση διάφορων μετρικών της επικοινωνίας CPU-μνήμης. Σε κάθε επανάληψη, χρησιμοποιούνται δύο νήματα, εκ των οποίων το ένα εκτελείται πάντα στον εικονικό πυρήνα 0, ενώ η εκτέλεση του άλλου νήματος γίνεται πρώτα στον εικονικό πυρήνα 4, ύστερα στον 8 και με το ίδιο βήμα καταλήγει στον εικονικό πυρήνα 60. Το συγκεκριμένο πρόγραμμα εμφανίζει στην οθόνη τον χρόνο που απαιτήθηκε για μερικές στοιχειώδεις πράξεις (πχ. ανάθεση σε μεταβλητή, κάποια πράξη κι ανάθεση κ.α.) που φανερώνουν την καθυστέρηση της επικοινωνίας των εικονικών πυρήνων με την μνήμη. Αν κάποια τιμή του χρόνου (σε δευτερόλεπτα) που επιστρέφεται μετά από κάθε επανάληψη του μετροπρογράμματος με διαφορετική ανάθεση νήματος σε λογικό πυρήνα είναι πολύ εξέχουσα, τότε μπορεί να υπήρχαν περισσότεροι `NUMA Nodes`. Ωστόσο δεν υπάρχουν, καθώς υπάρχει ομοιομορφία μεταξύ των τιμών αυτών.

Ο ακριβής τρόπος με τον οποίον επιτυγχάνεται η ανάθεση ενός νήματος σε έναν λογικό πυρήνα γίνεται σαφής με την ανάλυση που ακολουθεί.

5.2 Processor Affinity

Αναφερόμαστε στην δυνατότητα *Processor affinity* που ρυθμίζεται μέσω των μεταβλητών περιβάλλοντος **OMP_PLACES** και **OMP_PROC_BIND**. Οι δύο αυτές μεταβλητές εντοπίζονται περιληπτικά στο δεύτερο Κεφάλαιο του κειμένου. Γενικά, ο ρόλος αυτών των μεταβλητών είναι η ρύθμιση της εν λόγω δυνατότητας, που ελέγχει την δέσμευση ή αποδέσμευση μιας διεργασίας ή νήματος σε έναν συγκεκριμένο επεξεργαστή (CPU) ή σε ομάδα επεξεργαστών, ώστε να εκτελείται μόνο σε καθορισμένους επεξεργαστές-πυρήνες (λογικούς ή μη). Οι ακριβείς ρυθμίσεις για κάθε ομάδα πειραμάτων που χρειαζόμαστε δίνονται στο επόμενο κεφάλαιο, ενώ στην παρούσα ενότητα δίνεται μια εξήγηση του τρόπου χρήσης αυτών.

1. **OMP_PLACES**: Καθορίζει αν τα νήματα OpenMP δεσμεύονται σε συγκεκριμένες θέσεις (places) και κατ'επέκταση με ποιον τρόπο. Τιμές: true ή false για ενεργοποίηση ή όχι της λειτουργίας αυτής και master, close, spread για πιο ακριβή καθορισμό εφόσον είναι ενεργή. Καθώς το OpenMP υιοθετεί το

⟨list⟩ ⊨ ⟨p-list⟩ | ⟨aname⟩
⟨p-list⟩ ⊨ ⟨p-interval⟩ | ⟨p-list⟩,⟨p-interval⟩
⟨p-interval⟩ ⊨ ⟨place⟩:⟨len⟩:⟨stride⟩ | ⟨place⟩:⟨len⟩ | ⟨place⟩ | !⟨place⟩
⟨place⟩ ⊨ {⟨res-list⟩}
⟨res-list⟩ ⊨ ⟨res-interval⟩ | ⟨res-list⟩,⟨res-interval⟩
⟨res-interval⟩ ⊨ ⟨res⟩:⟨num-places⟩:⟨stride⟩ | ⟨res⟩:⟨num-places⟩ | ⟨res⟩ | !⟨res⟩
⟨aname⟩ ⊨ ⟨word⟩(⟨num-places⟩) | ⟨word⟩
⟨word⟩ ⊨ sockets | cores | threads | <implementation-defined abstract name>
⟨res⟩ ⊨ *non-negative integer*
⟨num-places⟩ ⊨ *positive integer*
⟨stride⟩ ⊨ *integer*
⟨len⟩ ⊨ *positive integer*

Σχήμα 5.1: Γραμματική ρύθμισης της OMP_PLACES

μοντέλο fork-join και υπάρχει ο ρόλος του γονέα-master νήματος, οι πολιτικές master, close και spread καθορίζουν συμβολικά την "απόσταση" μεταξύ των τοποθεσιών (places) εκτέλεσης των νημάτων μεταξύ του γονέα νήματος και των παιδιών του.

2. **OMP_PROC_BIND**: Καθορίζει τη λίστα θέσεων (places) ή το ποιες είναι η υποψήφιες θέσεις (cores, sockets ή threads) για την εκτέλεση OpenMP νημάτων. Για την λεπτομερή ρύθμισή της τηρείται η γραμματική του σχήματος 5.1. Γενικά, για τον ακριβή προσδιορισμό χρησιμοποιούνται λίστες με αριθμούς που αντιπροσωπεύουν τους δείκτες πυρήνων στους οποίους γίνεται αναφορά στο παρόν κεφάλαιο. Βεβαίως, όπως φαίνεται στην γραμματική, επιτρέπονται συμπτύξεις. Για παράδειγμα, η λίστα "{0},{2},{4},{6},{8},{10},{12},{14}" είναι ισοδύναμη με την "{0:15:2}" όταν ρυθμίζουμε την `OMP_PROC_BIND`.

Για τους σκοπούς των πειραμάτων μας, η `OMP_PROC_BIND` αρκεί να είναι ρυθμισμένη στην πολιτική *true*, σύμφωνα με την οποία πρέπει να τηρείται η κατανομή για μια ομάδα T νημάτων μεταξύ των P διαθέσιμων θέσεων (places) που ορίζουμε εμείς. Η ρύθμιση αυτή συνδυάζεται με την ρύθμιση της `OMP_PLACES` ανά πείραμα και ποικίλλει ανάλογα σε ποιους πυρήνες θέλουμε να εκτελεστουν οι επαναλήψεις του κάθε πειράματος. Για παράδειγμα, αν η `OMP_PLACES` έχει ρυθμιστεί ως {0:16:4} και η πολιτική της `OMP_PROC_BIND` είναι spread ενώ έχουν δημιουργηθεί 16 νήματα για εκτέλεση μιας εργασίας, τότε το σύστημα θα αναθέσει ένα νήμα σε κάθε έναν λογικό πυρήνα από τους δύο διαθέσιμους του κάθε φυσικού πυρήνα του `NUMA node` 0. Σε τελική ανάλυση, έτσι αναθέτουμε ένα νήμα σε κάθε φυσικό πυρήνα.

5.3 Μεθοδολογία και Προγραμματισμός Εκτέλεσης των Πειραμάτων

Σκοπός των πειραμάτων των οποίων τα αποτελέσματα παρατίθενται στην ενότητα αυτή είναι η εκτίμηση της βελτίωσης επιδόσεων που προκύπτει από τη χρήση των `futexes`. Ο βασικός τρόπος επίτευξης του σκοπού αυτού είναι η σύγκριση των επιδόσεων της εκτέλεσης κάποιου παράλληλου προγράμματος που χρησιμοποιεί κλήσεις `pthreads` για τον συγχρονισμό νημάτων και του ίδιου προγράμματος στο οποίο οι εν λόγω κλήσεις έχουν αντικατασταθεί με αντίστοιχες που χρησιμοποιούν `futexes`. Και στις δύο περιπτώσεις, το πρόγραμμα προς εκτέλεση προκύπτει από μετάφραση του OMPi, του *τροποποιημένου* που χρησιμοποιεί τις προτεινόμενες από την μελέτη αλλαγές και του - που αποτελεί τον κανονικό OMPi που χρησιμοποιεί pthreads.

Προκειμένου να επιτύχουμε τον σκοπό αυτόν, η μελέτη κατευθύνεται στην μέτρηση των επιδόσεων της εκτέλεσης μιας σουίτας `microbenchmarks` και μιας σουίτας εφαρμογών. Για την ακρίβεια, η πρώτη σουίτα ονομάζεται `EPCC OpenMP Microbenchmarks` που έχει συνταχθεί στο `Edinburgh Parallel Computing Center` και η δεύτερη `NAS Parallel Benchmarks`. Ακολουθεί μια περιγραφή του τρόπου με τον οποίον οι σουίτες αυτές χρησιμοποιήθηκαν.

5.3.1 EPCC OpenMP Microbenchmarks

Χρησιμοποιήθηκαν μερικές από τις συναρτήσεις του προγράμματος `syncbench.c` που διατίθεται, της οποίας οι συναρτήσεις εξετάζουν όλα τα constructs του OpenMP. Η μελέτη σχετίζεται με την εκτέλεση συγκεκριμένων συναρτήσεων του εν λόγω προγράμματος, καθεμία από τις οποίες εξετάζουν μια λειτουργία του OpenMP σε μια δομή επανάληψης `for` εντός μιας παράλληλης περιοχής. Κάθε εκτέλεση της συνάρτησης αποτελεί πράγματι ένα πείραμα κι αντιστοιχεί και σε μια ονομασία που χρησιμοποιείται στα πλαίσια της εργασίας. Οι πιο χρήσιμες συναρτήσεις κι εκείνες που μπορούν να αναδείξουν την διαφορά στις επιδόσεις μέσω της χρήσης futexes είναι οι εξής:

- **Πείραμα CRITICAL**: Με τη συνάρτηση testcrit() υλοποιείται μια κρίσιμη περιοχή ανά βρόχο, η οποία ορίζεται με την οδηγία `#pragma omp critical`.

- **Πείραμα LOCK/UNLOCK**: Με τη συνάρτηση testlock() υλοποιείται μια κρίσιμη περιοχή ανά βρόχο, η οποία οριοθετείται με τις OpenMP συναρτήσεις `omp_set_lock()` και `omp_unset_lock()`.

- **Πείραμα BARRIER**: Με τη συνάρτηση testbar() υλοποιείται ένα φράγμα (`barrier`) μετά την εκτέλεση των βρόχων, με την οδηγία `#pragma omp barrier`.

Η ιδέα είναι πως η βασική συνάρτηση `benchmark()` καλεί τις επιλεγμένες αυτές συναρτήσεις στην αρχή του κώδικα του `syncbench.c` και πραγματοποιεί τις χρονομετρήσεις, οι οποίες τελικά επιστρέφονται ως έξοδος του προγράμματος αυτού, μαζί με διάφορα στατιστικά (outliers, Standard Deviation κ.α.). Η μοναδική μετρική που μας αφορά και χρησιμοποιείται στα γραφήματα είναι η `overhead` ή αλλιώς ο χρόνος εκείνος που δεν αντιστοιχεί στην συνολική εκτέλεση του προγράμματος αλλά σε εκείνον που σπαταλάται στον συγχρονισμό ή και στη δρομολόγηση των νημάτων κατά την εκτέλεσή του.

Βασικό προαπαιτούμενο για την εκτέλεση των πειραμάτων αποτελεί μια κανονική πρόσφατη έκδοση του `OMPi` και η ίδια έκδοση του `OMPi` η οποία φέρει τις τροποποιήσεις που συζητήθηκαν. Η ουσία των πειραμάτων είναι η σύγκριση των επιδόσεων εκτέλεσης του προγράμματος `syncbench.c` με τις κατάλληλες ρυθμίσεις για την επιλογή των συναρτήσεων, μεταξύ των εκτελέσιμων αρχείων που προκύπτουν για αυτό από την τροποποιημένη έκδοση του `OMPi` και την μη-τροποποιημένη. Για περεταίρω ανάλυση, στις συγκρίσεις συμμετέχει και η εκτέλεση που προκύπτει από την μετάφραση των προγραμμάτων από τον γνωστό `gcc`.

5.3.2 Σουίτα εφαρμογών NAS

Τα NAS Parallel Benchmarks (*NASA Advanced Supercomputing Division Parallel Benchmarks* ή NPB) είναι ένα μικρό σύνολο προγραμμάτων σχεδιασμένων για την αξιολόγηση της απόδοσης παράλληλων υπερυπολογιστών. Αρχικά προέρχονταν από εφαρμογές για ρευστομηχανική (Computational Fluid Dynamics CFD). Στη συνέχεια, η συλλογή επεκτάθηκε ώστε να περιλαμβάνει benchmarks για μη-δομημένα adaptive meshes, παράλληλη εισαγωγή/εξαγωγή δεδομένων (I/O), multi-zone εφαρμογές και computational grids. Τα μεγέθη των προβλημάτων είναι προκαθορισμένα και ταξινομούνται σε διαφορετικές κλάσεις. Υπάρχουν reference implementations διαθέσιμες σε δημοφιλή προγραμματιστικά μοντέλα, όπως MPI και OpenMP (NPB 2

και NPB 3), που χρησιμοποιούνται ευρέως για benchmarking και πειράματα παραλληλισμού [12]. Οι περισσότερες εφαρμογές είναι γραμμένες με τη γλώσσα Fortran.

Αργότερα, το Πανεπιστήμιο της Tsukuba φρόντισε για την μετατροπή του πηγαίου κώδικα των προγραμμάτων από Fortran σε C, καθώς και για την παραλληλοποίησή του, όταν ανέπτυσσε τον μεταφραστή `omni`, έναν από τους πρώτους ερευνητικούς compilers για OpenMP.

Από τις διαθέσιμες εφαρμογές, για τους σκοπούς της μελέτης χρησιμοποιήθηκαν οι εξής:

1. **MG – Multi-Grid**: Πρόκειται για τον υπολογισμό της λύσης μιας τρισδιάστατης διακριτής εξίσωσης Poisson.

2. **LU - Lower-Upper Gauss-Seidel solver**: Πρόκειται για τον υπολογισμό λύσης συστήματος γραμμικών εξισώσεων ρευστομηχανικής.

3. **EP - Embarissingly Parallel**: Αφορά την εκτίμηση της απόδοσης του παραλληλισμού ενός συστήματος, μέσω της επίλυσης ολοκληρωτικά ανεξάρτητων μεταξύ τους προβλημάτων.

Εκτός από τις εφαρμογές, σημαντικό είναι να αναφέρουμε το πλήθος επιλογών του μεγέθους του προβλήματος ή αλλιώς κλάσεων.

1. Τάξη S: μικρή, για γρήγορες δοκιμές

2. Τάξη W: μέγεθος workstation (σταθμός εργασίας της δεκαετίας του '90· τώρα πιθανώς πολύ μικρή)

3. Τάξεις A, B, C: τυπικά προβλήματα δοκιμής; περίπου 4 φορές μεγαλύτερα κατά τη μετάβαση από την μία τάξη στην επόμενη

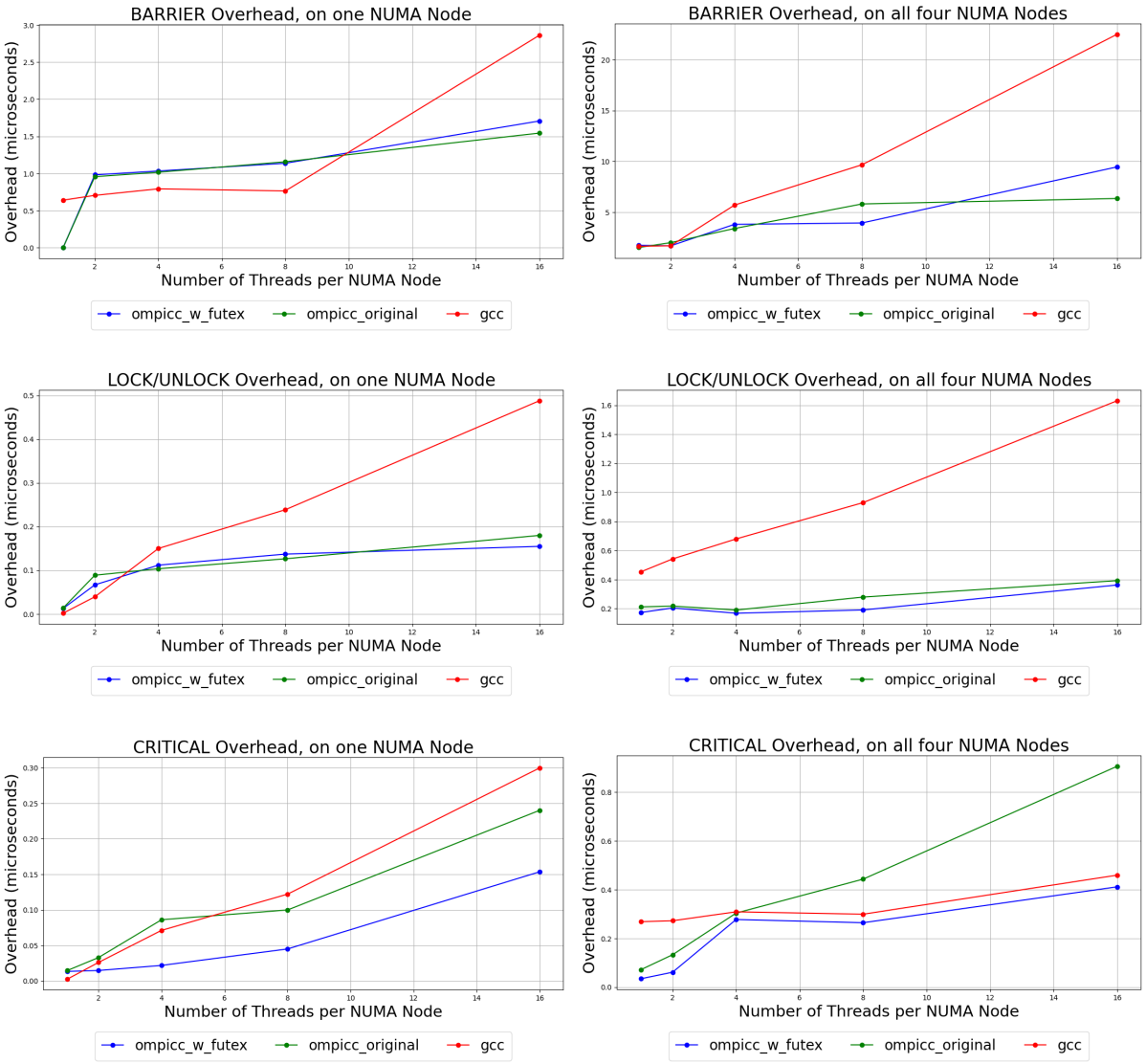
4. Τάξεις D, E, F: μεγάλα προβλήματα δοκιμής; περίπου 16 φορές μεγαλύτερα από την προηγούμενη τάξη

Για τους σκοπούς της μελέτης και για τη λήψη αξιόπιστων μετρήσεων, έγινε χρήση της κλάσης B και έγινε η εκτέλεση κάθε προγράμματος για κάθε διαθέσιμο μεταφραστή (gcc, τροποποιημένο OMPi για τη χρήση futexes και μη-τροποποιημένο OMPi).

5.4 Αποτελέσματα Πειραμάτων EPCC OpenMP Microbenchmarks

Στα παρακάτω γραφήματα (σχ. 5.2), φαίνεται πώς διαμορφώνεται ο χρόνος επιβάρυνσης που σπαταλάται σε σε σχέση με το πλήθος νημάτων ανά NUMA node. Η μονάδα μέτρησης που χρησιμοποιείται για την μέτρηση του χρόνου αυτού είναι τα microseconds (μs ή 10^{-6} δευτερόλεπτα). Το μέγεθος που περιγράφεται είναι ο χρόνος που σπαταλάται για την δρομολόγηση και τον συγχρονισμό των νημάτων. Σε κάθε γράφημα, απεικονίζονται τρεις καμπύλες όπου κάθε μια από αυτές αντιστοιχεί στις επιδόσεις ενός μεταφραστή: η μπλε χρώματος για την έκδοση του OMPi που χρησιμοποιεί futexes όπως έχει περιγραφτεί, η πράσινου χρώματος για την κανονική έκδοση του OMPi και η κόκκινου χρώματος για τον gcc.

Η εκτέλεση του πειράματος απαιτούσε την εκτέλεση πολλαπλών επαναλήψεων. Σε κάθε μια υπάρχει διαφορετικό πλήθος νημάτων και τοποθετημένα διαφορετικούς πυρήνες. Ζητούμενη ήταν η σωστή ρύθμιση των σχετικών μεταβλητών OMP_PROC_BIND και OMP_PLACES, εκτός της OMP_NUM_THREADS. Και οι τρείς έχουν συζητηθεί. Χρησιμοποιούνται είτε ένας είτε και οι τέσσερις διαθέσιμοι κόμβοι. Σε κάθε επανάληψη καθορίζουμε στο σύστημα το πλήθος των νημάτων (1,2,4,8,16 ανά κόμβο), την πολιτική ανάθεσης (OMP_PROC_BIND=`true`) και οι θέσεις/πυρήνες ανάθεσης (ποικίλλουν, βλ. πιν. 5.3). Το πλήθος νημάτων ανά επανάληψη φαίνεται στον άξονα x των παρακάτω γραφημάτων. Παράλληλα, στον άξονα y, αντιστοιχούν οι χρόνοι επιβάρυνσης που προκύπτουν από την εκτέλεση κάθε πειράματος.



Σχήμα 5.2: Αποτελέσματα πειραμάτων για τα πειράματα **BARRIER**, **LOCK/UNLOCK** και **CRITICAL** εκτελεσμένα σε 1 και 4 NUMA Nodes.

Επίσης, το σκεπτικό ανάθεσης νημάτων στους πυρήνες των κόμβων ακολουθεί μια ομοιομορφία. Για παράδειγμα, όταν χρησιμοποιούμε οκτώ νήματα και τέσσερις κόμβους αναθέτουμε δύο νήματα στον κάθε έναν κόμβο. Φροντίσαμε ώστε να υπάρχει ίσο πλήθος νημάτων μεταξύ των τεσσάρων κόμβων και αυτή η πολιτική ισχύει και στην επόμενη σειρά πειραμάτων.

Κοιτώντας τα γραφήματα αντιλαμβανόμαστε πως σε κάθε πείραμα οι επιδόσεις της τροποποιημένης έκδοσης του OMPi συγκρίνεται άμεσα με αυτές του μητροποποιημένου OMPi. Επίσης, οι επιδόσεις του gcc γίνονται μάλλον χειρότερες σε σχέση με τις δύο εκδόσεις του OMPi καθώς αυξάνεται ο παραλληλισμός, αφού ο OMPi αποδεικνύεται βέλτιστος για τα παράλληλα προγράμματα όπως αυτά που

εξετάζονται.

Στην πλειονότητα των περιπτώσεων, παρατηρούμε εύκολα πως η καμπύλη της μη-τροποποιημένης έκδοσης του OMPi δεν έχει μεγάλη απόσταση από εκείνη της αντίστοιχης τροποποιημένης. Σε αυτές τις περιπτώσεις, η εικόνα των αντίστοιχων καμπυλών πλησιάζει τα όρια του στατιστικού σφάλματος.

Το πείραμα στο οποίο η επίδοση του τροποποιημένου OMPi ξεχωρίζει είναι το επονομαζόμενο CRITICAL. Η αιτία δεν έγινε σαφής κατά την διάρκεια της μελέτης, καθώς εν τέλει ο OMPi αντικαθιστά την οδηγία `#pragma omp critical` με συναρτήσεις κλειδώματος-ξεκλειδώματος και η εκτέλεση μοιάζει τελικά με κλασικό κλείδωμα-ξεκλείδωμα μιας κρίσιμης περιοχής.

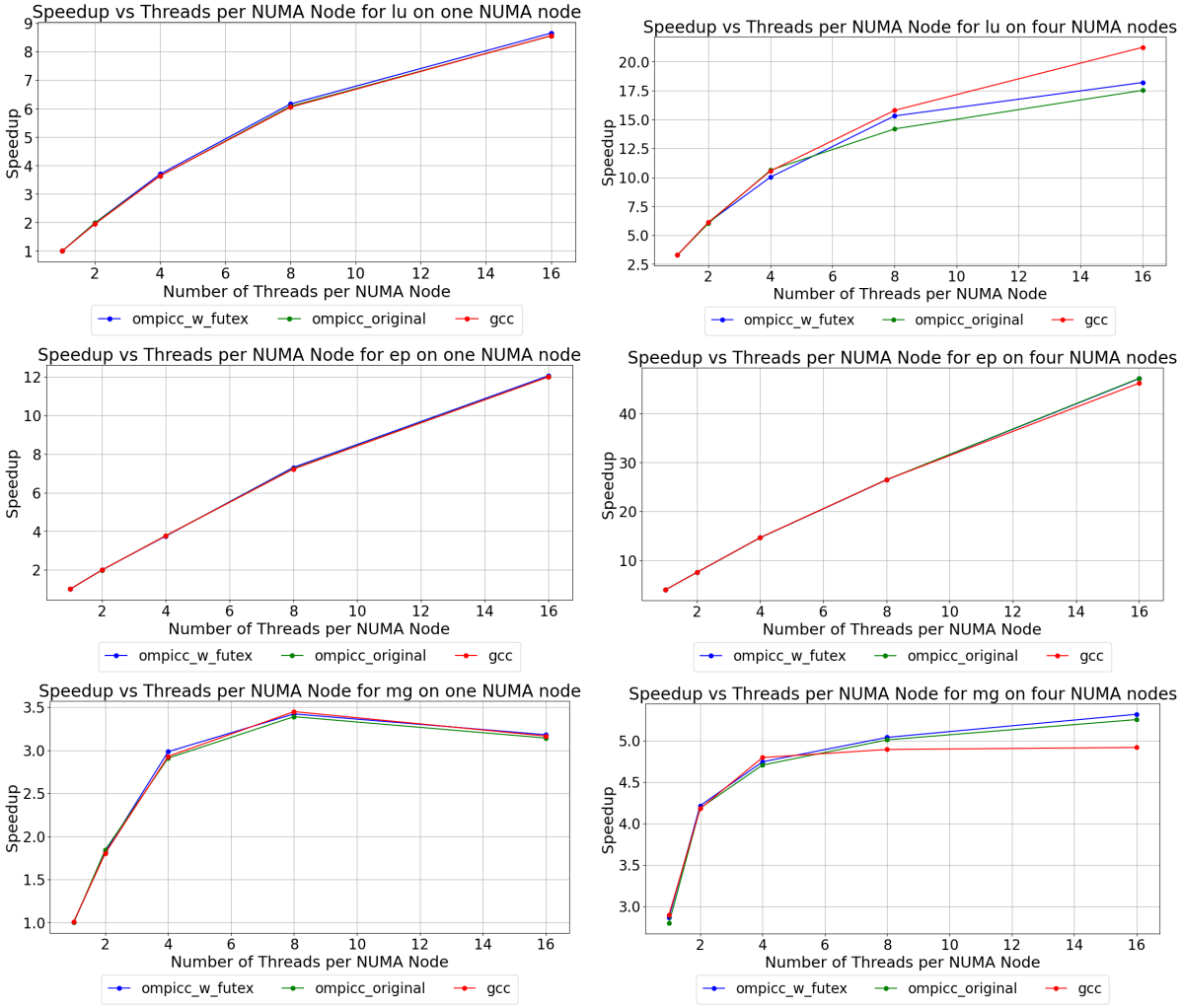
5.5 Αποτελέσματα Πειραμάτων NAS

Στα παρακάτω γραφήματα (σχ. 5.3), φαίνεται πώς διαμορφώνεται η μετρική *speedup* σε σε σχέση με το πλήθος νημάτων ανά NUMA node. Με τον όρο speedup εννοούμε το κλάσμα μεταξύ του συνολικού χρόνου σειριακής εκτέλεσης (ή αλλιώς με ένα νήμα) T_s και του συνολικού χρόνου παράλληλης εκτέλεσης T_n για ένα πλήθος νημάτων. Τελικά, η τιμή της μετρικής αυτή υπολογίζεται με τον τύπο: $\frac{T_s}{T_n}$. Όπως και στην προηγούμενη υποενότητα, σε κάθε γράφημα, απεικονίζονται τρεις καμπύλες όπου κάθε μια από αυτές αντιστοιχεί στις επιδόσεις ενός μεταφραστή: η μπλε χρώματος για την έκδοση του OMPi που χρησιμοποιεί futexes όπως έχει περιγραφτεί, η πράσινου χρώματος για την κανονική έκδοση του OMPi και η κόκκινου χρώματος για τον gcc.

Παρομοίως με την προηγούμενη σειρά πειραμάτων, η σωστή ρύθμιση των σχετικών μεταβλητών OMP_PROC_BIND, OMP_PLACES και OMP_NUM_THREADS αποτελούσε ζητούμενο. Ακολουθήθηκε η ίδια πολιτική για την OMP_PROC_BIND και χρησιμοποιούνται ξανά είτε ένας είτε και οι τέσσερις διαθέσιμοι κόμβοι. Ο άξονας x εξακολουθεί να αντιστοιχεί στο πλήθος νημάτων κάθε επανάληψης και οι διατάξεις της εκτέλεσης κάθε πειράματος είναι η ίδια (1,2,4,8,16 ανά κόμβο). Η ουσιαστική διαφορά με την προηγούμενη σειρά πειραμάτων είναι η σημασία του άξονα y, που αντιστοιχεί στην μετρική speedup, η οποία δεν συνοδεύεται από κάποια μονάδα μέτρησης. Αναπαριστά μόνο την διαφορά στην αύξηση των επιδόσεων.

Σημαντικό είναι να αναφέρουμε πως σε κάθε καμπύλη που απεικονίζεται στα

παρακάτω γραφήματα, ο σειριακός χρόνος εκτέλεσης σε κάθε υπολογισμό του speedup, είναι εκείνος του αντίστοιχου μεταφραστή. Με άλλα λόγια, η καμπύλη κάθε μεταφραστή σχηματίζεται από υπολογισμούς speedup για χρόνους εκτέλεσης που αντιστοιχούν αποκλειστικά στο πρόγραμμα που είναι μεταφρασμένο από αυτόν.



Σχήμα 5.3: Αποτελέσματα πειραμάτων για τα πειράματα **LU**, **MG** και **EP** εκτελεσμένα σε 1 και 4 NUMA Nodes.

Συνολικά, οι επιδόσεις του τροποποιημένου OMPi δεν διαφέρουν σημαντικά από εκείνη του μη-τροποποιημένου OMPi σύμφωνα με τα παραπάνω γραφήματα. Παρατηρούμε κιόλας πως σε όποια περίπτωση ο τροποποιημένος OMPi δεν έχει μεγαλύτερη τιμή speedup από εκείνη του μη-τροποποιημένου, οι αντίστοιχες τιμές τους σχεδόν ταυτίζονται.

5.6 Τελικά συμπεράσματα

Αυτό που μπορούμε να συμπεράνουμε με ασφάλεια είναι πως η χρήση των futexes σίγουρα μπορεί να συνεισφέρει στην βελτίωση των επιδόσεων εκτέλεσης παράλληλων εφαρμογών, καθώς η υλοποίηση των συναρτήσεων κλειδώματος και ξεκλειδώματος απαιτούν πολύ λιγότερες γραμμές κώδικα για να πετύχουμε τον συγχρονισμό των νημάτων σε σχέση με τις αντίστοιχες που εκτελούνται από την pthreads. Δείξαμε πως είναι δυνατό η χρήση των futexes για τον συγχρονισμό νημάτων μπορεί να οδηγήσει σε καλύτερες επιδόσεις στην εκτέλεση ορισμένων προγραμμάτων και πως η αποκλειστική χρήση τους αντί της καθιερωμένης χρήσης των pthreads αποδεικνύεται συχνά ωφέλιμη.

Εμπόδιο όμως μπορεί να σταθεί η τοποθέτηση της μεταβλητής futex στη μνήμη όταν η αρχιτεκτονική του συστήματος είναι NUMA, όπως στην περίπτωσή μας. Στην γενική περίπτωση, δεν υπάρχει ξεκάθαρος τρόπος της τοποθέτησης της μεταβλητής αυτής στη μνήμη και κατ'επέκταση δεν μπορούμε να γνωρίζουμε σε ποιου επεξεργαστή τη μνήμη είναι αποθηκευμένη. Μια ιδέα θα ήταν η δημιουργία ενός κατανεμημένου συστήματος κλειδώματος/ξεκλειδώματος NUMA που χρησιμοποιεί futexes. Η τοποθέτηση ενός futex σε κάθε κόμβο NUMA πιθανόν να αποτελούσε ουσιαστική συμβολή στην μείωση της καθυστέρησης της επικοινωνίας μεταξύ των κόμβων.

Σαν τελικό σχόλιο, αναφέρουμε πως στην εκτέλεση των πειραμάτων έχει σημαντικό ρόλο και το περιβάλλον εκτέλεσης και ο εκάστοτε υπάρχων φόρτος εργασίας του κατά την εκτέλεση των πειραμάτων. Είναι πιθανό ότι σε ένα διαφορετικό σύστημα, όπως ένα SMP αντί για NUMA, και με μικρότερο συνολικό φόρτο, όπως συμβαίνει στα μηχανήματα του PPG, θα παρατηρούσαμε διαφορετικά —αν και πιθανότατα παρόμοια— αποτελέσματα. Εξάλλου, ο συγχρονισμός νημάτων σπάνια αντιπροσωπεύει το μεγαλύτερο ποσοστό του εκτελούμενου κώδικα σε ένα πρόγραμμα.

Κεφαλαιο 6

Μελλοντικες Προεκτασεις κι Επιλογος

Η παρούσα Διπλωματική Εργασία ολοκληρώνεται με τα περιεχόμενα του παρόντος κεφαλαίου. Δίνονται μερικές ιδέες για τον μελλοντικό ερευνητή αποδοτικών, αξιόπιστων και φορητών μεθόδων συγχρονισμού νημάτων, καθώς και μερικά καταληκτικά σχόλια.

6.1 Μελλοντικές Προεκτάσεις

Αν και στις σελίδες της εργασίας αυτής έχει γίνει μια εκτενής μελέτη της σημασίας και των οφελών της χρήσης του ευέλικτου αλλά περίπλοκου μηχανισμού των futexes, η μελέτη των futexes είναι ατέρμονη και πάντα επιδέχεται επεκτάσεις. Ακολουθούν μερικές προτάσεις περαιτέρω έρευνας:

- Δημιουργία βιβλιοθήκης πολυνηματισμού που στοχεύει κυρίως σε επιδόσεις και χρησιμοποιεί futexes.
- Δημιουργία ολοκληρωμένης EELIB για τον OMPi που χρησιμοποιεί αποκλειστικά ατομικές λειτουργίες, futexes αλλά και διάφορες προηγμένες λειτουργίες αυτών για τον συγχρονισμό νημάτων ή/και διεργασιών. Ένα σχετικό πα-

ράδειγμα προηγμένης λειτουργίας αποτελούν τα λεγόμενα *Private Futexes*, τα οποία αφορούν τον συγχρονισμό νημάτων μίας και μόνο διεργασίας.

- Δημιουργία ενός κατανεμημένου συστήματος κλειδώματος/ξεκλειδώματος NUMA που χρησιμοποιεί futexes.

6.2 Τελικά σχόλια

Η μελέτη των futexes αποτέλεσε μια ενδιαφέρουσα δραστηριότητα για το Parallel Processing Group του Πανεπιστημίου Ιωαννίνων εξαιτίας της ερευνητικής αξίας που εμπεριέχει και της πιθανής χρησιμότητάς του στις άλλες δραστηριότητές του. Απώτερος σκοπός της αποτελεί η διάδοση της ιδέας στην οποία στηρίζεται η συνήθης πρακτική όσον αφορά τον συγχρονισμό νημάτων και ταυτόχρονα η περαιτέρω εδραίωση της θέσης του μηχανισμού στον τομέα του παράλληλου προγραμματισμού μέσα από τα πειράματα που προηγήθηκαν.

Βιβλιογραφια

[1] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," 2002.

[2] futex(2) - Linux manual page. Accessed: 2025-06-15. [Online]. Available: https://man7.org/linux/man-pages/man2/futex.2.html

[3] U. Drepper and I. Molnár, "The native posix thread library for linux," p. 3, Feb. 2003, accessed: 2025-06-17. [Online]. Available: https://www.cs.utexas.edu/~witchel/372/lectures/POSIX_Linux_Threading.pdf

[4] Microsoft Corporation. Waitonaddress function (synchapi.h). Accessed: 2025-06-17. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitonaddress

[5] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable c compiler for openmp v.2.0," 2003.

[6] G. C. Philos, V. V. Dimakopoulos, and P. E. Hadjidoukas, "A runtime system architecture for ubiquitous support of openmp," 2008.

[7] "Openmp execution model," https://www.openmp.org/spec-html/5.0/openmpse3.html, accessed: 2025-06-22.

[8] U. Drepper, "Futexes are tricky," pp. 6–7, 2004, https://www.akkadia.org/drepper/futex.pdf.

[9] R. Denis-Courmont. Condition variables using futexes. Accessed: 2025-06-29. [Online]. Available: https://www.remlab.net/op/futex-condvar.shtml

[10] GNU, "Atomic Memory Model Synchronization - GCC Wiki," https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync, accessed: 2025-06-29.

[11] J. D. McCalpin. (1995) Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/. Original benchmark source and documentation.

[12] NASA Advanced Supercomputing (NAS) Division, "Nas parallel benchmarks," https://www.nas.nasa.gov/software/npb.html, 2025, accessed: 2025-10-20. [Online]. Available: https://www.nas.nasa.gov/software/npb.html

Παραρτημα Α

Παραρτημα Κωδικα

```
#!/bin/bash

# Compile stream.c
echo "Compiling stream.c..."
gcc -O -fopenmp stream.c -o stream_omp
if [[ $? -ne 0 ]]; then
  echo "Compilation failed. Exiting."
  exit 1
fi
echo "Compilation successful."

# Set number of OpenMP threads and binding policy
export OMP_NUM_THREADS=2
export OMP_PROC_BIND=spread

# Define PUs for each core on Socket 0 (physical cores)
PS_CORE_0=(0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60)

# Number of cores
NUM_CORES=${#PS_CORE_0[@]}

# Iterate over second thread's PU
for ((i=1; i<NUM_CORES; i++)); do
```

```
  THREAD0_P=${PS_CORE_0[0]}   # Fixed
  THREAD1_P=${PS_CORE_0[$i]}  # Varying

  export OMP_PLACES="{${THREAD0_P}},{${THREAD1_P}}"

  echo "================================================="
  echo "Running with:"
  echo "OMP_NUM_THREADS = $OMP_NUM_THREADS"
  echo "OMP_PROC_BIND   = $OMP_PROC_BIND"
  echo "OMP_PLACES      = $OMP_PLACES"

  ./stream_omp | awk '
    /Number of Threads requested/ { print }
    /Number of Threads counted/   { print }
    /Copy :|Scale :|Add :|Triad :/ { printf "%s Avg time: %s\n", $1,
       $3 }'

  echo ""
done
```

Κώδικας Α.1: Script για την Εκτίμηση χρόνου επικοινωνίας μεταξύ μνήμης και εικονικών πυρήνων με τη χρήση του stream benchmark

```
typedef struct {                /* For nested locks */
    uint32_t lock;              /* The lock in question */
    uint32_t ilock;             /* Lock to access the whole struct */
    uint32_t cond;              /* For waiting until lock is
       available */
    int count;                  /* # times locked by the same thread
       */
    void *owner;                /* The owner (task) of the nestable
       lock */
} othr_nestlock_t;

typedef union {
    struct {
        int type;               /* normal/spin/nested */
```

```
        union {
            uint32_t normal;          /* normal lock */
            othr_nestlock_t nest;     /* nest lock */
            struct {
                int rndelay;          /* Used for initial spin
                   delays */
                uint32_t mutex;
            } spin;
        } data;
    } lock;
    char padding[CACHE_LINE];
} othr_lock_t;
```

Κώδικας Α.2: Τροποποιημένες δομές othr_lock_t και othr_nestlock_t

```
int othr_init_lock(othr_lock_t *lock, int type)
{
  switch (lock->lock.type = type)
  {
    case ORT_LOCK_NEST:
    {
      othr_nestlock_t *l = &(lock->lock.data.nest);

      l->ilock = 0; /* 0 is for available. 1 is for 1 waiter and 2
         for at least 1 waiter on a lock */
      l->lock  = 0;
      l->count = 0;
      l->cond  = 0;
      return (0);
    }

    case ORT_LOCK_SPIN:
    {
      lock->lock.data.spin.rndelay = 0;
      lock->lock.data.spin.mutex = 0;
      return (0);
    }
```

```
22
23      default: /* ORT_LOCK_NORMAL */
24        lock->lock.data.normal = 0;
25        return (0);
26    }
27  }
```

Κώδικας Α.3: Τροποποιημένη othr_init_lock()

```
1  int othr_destroy_lock(othr_lock_t *lock)
2  {
3    return (0);
4  }
```

Κώδικας Α.4: Τροποποιημένη othr_destroy_lock()

```
1  int othr_set_lock(othr_lock_t *lock)
2  {
3    switch (lock->lock.type)
4    {
5      case ORT_LOCK_NEST:
6      {
7        othr_nestlock_t *l = &(lock->lock.data.nest);
8        void            *me = ort_get_current_task();
9
10       flock(&l->ilock);
11       if (cmpxchg(&l->lock,0,1)) /* If not locked, lock it */
12       {
13         l->owner = me;                          /* Get ownership */
14         l->count++;
15       }
16       else
17       {
18         if (l->owner == me)                     /* Did i do that? */
19           l->count++;
20         else                                    /* Locked by someone
               else */
21         {
```

```
22            while (!cmpxchg(&l->lock,0,1))    /* Go to sleep while the
                 lock is taken */
23              fwait(&l->cond, &l->ilock);
24            l->owner = me;
25            l->count++;
26          }
27        }
28        funlock(&l->ilock);
29        return (0);
30      }
31
32      case ORT_LOCK_SPIN:
33      {
34        /* General, portable solution: spin trying with exponential
             backoff */
35        volatile int count, delay, dummy;
36        for (delay = lock->lock.data.spin.rndelay;
37             !cmpxchg(&(lock->lock.data.spin.mutex),0,1);)
38        {
39          for (count = dummy = 0; count < delay; count++)
40            dummy += count;       /* To avoid compiler optimizations */
41          if (delay == 0)
42            delay = 1;
43          else
44            if (delay < 10000)      /* Don't delay too much */
45              delay = delay << 1;
46        }
47        lock->lock.data.spin.rndelay++;    /* Next thread would wait a
             bit more */
48        return (0);
49      }
50      default: /* ORT_LOCK_NORMAL */
51        return flock(&(lock->lock.data.normal));
52    }
53  }
```

Κώδικας Α.5: Τροποποιημένη othr_set_lock()

```
int othr_unset_lock(othr_lock_t *lock)
{
  switch (lock->lock.type)
  {
    case ORT_LOCK_NEST:
    {
      othr_nestlock_t *l = &(lock->lock.data.nest);

      flock(&l->ilock);
      if (l->owner == ort_get_current_task() && l->count > 0)
      {
        l->count--;
        if (l->count == 0)
        {
          funlock(&l->lock);
          fsignal(&l->cond);
        }
      }
      funlock(&l->ilock);
      return 0;
    }

    case ORT_LOCK_SPIN:

      lock->lock.data.spin.rndelay = 0;    /* reset it */
      return funlock(&(lock->lock.data.spin.mutex));

    default: /* ORT_LOCK_NORMAL */
      return funlock(&(lock->lock.data.normal));
  }
}
```

Κώδικας Α.6: Τροποποιημένη othr_unset_lock()

```
int othr_test_lock(othr_lock_t *lock)
{
  switch (lock->lock.type)
  {
    case ORT_LOCK_NEST:
    {
      othr_nestlock_t *l = &(lock->lock.data.nest);
      int res;

      flock(&l->ilock);
      if (cmpxchg((&l->lock),0,1))  /* If the lock is free, set new
          owner */
      {
        l->owner = ort_get_current_task();
        res = ++l->count;
      }
      else
        if (l->owner == ort_get_current_task())
          res = ++l->count;
        else
          res = 0;
      funlock(&l->ilock);
      return res;
    }

    case ORT_LOCK_SPIN:
      return (cmpxchg(&(lock->lock.data.spin.mutex),0,1));

    default: /* ORT_LOCK_NORMAL */
      return (cmpxchg(&(lock->lock.data.normal),0,1));
  }
}
```

Κώδικας A.7: Τροποποιημένη othr_test_lock()