Πτυχιακή Εργασία

Ανδρέας Ανδρέου

Ιούλιος 2008

Ατομικές Λειτουργίες & Διεργασίες στον παραλληλοποιητικό μεταφραστή OMPi

Επιβλέπων:

Βασίλειος Δημακόπουλος

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

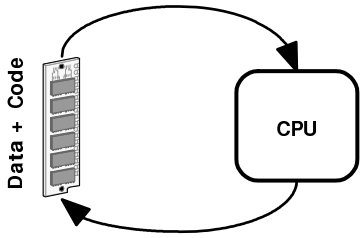
1.1 Υπολογιστές & Εξέλιξη

Οι πρώτοι ηλεκτρονικοί υπολογιστές έκαναν την εμφάνιση τους την δεκαετία του 1940. Την τότε εποχή ο κορυφαίος υπολογιστής σε επιδόσεις ήταν ο ENIAC. Ο υπολογιστής αυτός καταλάμβανε γύρο στα 63τ.μ χώρο ήταν ικανός να εκτελέσει 5000 ακέραιες προσθέσεις το δευτερόλεπτο. Ένας απλός υπολογιστής σήμερα, με αμελητέο μέγεθος μπορεί να εκτελέσει δισεκατομμύρια πράξεις με πραγματικούς αριθμούς στο ίδιο χρονικό διάστημα.

Θα μπορούσε κάποιος να ισχυριστεί ότι με τέτοια δραματική εξέλιξη, οι υπολογιστές, σε μερικά χρόνια θα μπορούσαν να επιλύσουν οποιοδήποτε υπολογίσιμο πρόβλημα, όσο περίπλοκο και να είναι σε μηδαμινό χρόνο. Αυτός ο ισχυρισμός ενδυναμώνεται και από το θεώρημα το Moore, που λέει ότι η ταχύτητα των υπολογιστών αυξάνεται σε χρονικό διάστημα μικρότερο των δύο χρόνων. Μήπως όμως υπάρχουν κάποια όρια στην καλπάζουσα αυτή άνοδο·

Η αύξηση της ταχύτητα των επεξεργαστών, όντως, δεν είναι δυνατόν να συνεχιστεί επ αόριστό. Φυσικοί νόμοι, όπως η ταχύτητα των ηλεκτρονίων, που δεν μπορεί να ξεπεράσει την ταχύτητα του φωτός, καθώς και οι περιορισμοί στην πυκνότητα των transistor ανά μονάδα επιφάνειας θέτουν ένα άνω όριο στην αύξηση αυτή της ταχύτητας των υπολογιστών. Όμως πολύπλοκα προβλήματα, όπως η πρόβλεψη του καιρού, και γενικά προβλήματα που πρέπει να λυθούν σε σύντομο χρονικό διάστημα αιωρούνται στο χώρο.

Τα παράλληλα συστήματα, φαίνεται τα είναι η λύση στα προβλήματα αυτά. Ως φυσική εξέλιξη του κλασσικού μοντέλου προγραμματισμού/αρχιτεκτονικής von Neumann ή αλλιώς σειριακό, έρχεται στο προσκήνιο το παράλληλο μοντέλο υπολογισμού και η παράλληλη επεξεργασία.



Σχήμα 1.1: Μοντέλο von Neumann

1.2 Παράλληλοι Υπολογιστές

Ο παράλληλος υπολογιστής μπορεί να οριστεί ως ένα σύνολο επεξεργαστών οι οποίοι είναι δυνατό να συνεργαστούν για την επίλυση ενός υπολογιστικού προβλήματος. Αφαιρετικά ένα παράλληλο σύστημα σπάει το πρόβλημα σε μικρότερα προβλήματα, τα επιλύει ταυτόχρονα (παράλληλα) και έπειτα συγκεντρώνει τα αποτελέσματα, εξάγοντας το τελικό. Με αυτό τον τρόπο αν έχουμε ένα παράλληλο υπολογιστή με N επεξεργαστές, θεωρητικά θα επιλύσουμε το πρόβλημα N φορές πιο γρήγορα σε σύγκριση με ένα σειριακό υπολογιστή με ένα επεξεργαστή.

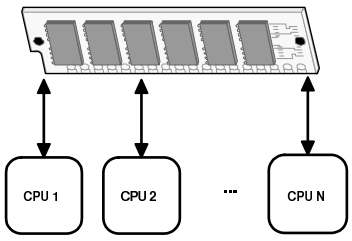
Στους παράλληλους υπολογιστές υπάρχουν γενικά δύο μοντέλα προγραμματισμού. Το μοντέλο κοινής μνήμης και το μοντέλο μεταβίβασης μηνυμάτων.

1.2.1 Μοντέλο Κοινής Μνήμης

Το μοντέλο κοινής μνήμης ή αλλιώς κοινού χώρου διευθύνσεων δεν υπάρχει διασύνδεση μεταξύ των επεξεργαστών, παρά μόνο ανάμεσα στους επεξεργαστές και τη μνήμη η οποία είναι άμεσα προσπελάσιμη από όλους. Κάθε επεξεργαστής μπορεί να εκτελεί διαφορετική εντολή επάνω σε διαφορετικά δεδομένα τα οποία όμως μπορούν να τα προσπελάσουν όλοι. Αυτό έχει ως αποτέλεσμα ότι όλοι οι επεξεργαστές "βλέπουν" την ίδια μνήμη, με τον ίδιο τρόπο. Προγραμματιστικά, η κατάσταση δεν διαφέρει πολύ από το γνώριμο σειριακό μοντέλο. Η επικοινωνία μεταξύ των επεξεργαστών γίνεται μέσω της τροποποίησης κοινών μεταβλητών στην μνήμη.

Ο πιο διαδεδομένος τρόπος προγραμματισμού σε αυτό το μοντέλο είναι η χρήση νημάτων. Μια διεργασία μπορεί να περιέχει ένα αριθμό νημάτων τα οποία διαμοιράζονται τον χώρο διευθύνσεών της. Τα νήματα ως «ελαφρές» διεργασίες εκτελούνται παράλληλα. Υπάρχουν αρκετές υλοποιήσεις νημάτων σε μορφή βιβλιοθήκης με πιο δημοφιλή τα POSIX και Solaris threads.

Σχετικά πρόσφατα ένας άλλος τρόπος προγραμματισμού στο μοντέλο κοινής μνήμης έχει αρχίσει να γίνεται ιδιαίτερα δημοφιλές, και ονομάζεται OpenMP. Το OpenMP είναι μια διεπαφή εφαρμογών προγραμμάτων API το οποίο με την προσθήκη ορισμένων εντολών σε κάποια σημεία του κώδικα μετατρέπει τον κώδικα σε παράλληλο. Το πρότυπο χαρακτηρίζεται τόσο για την ευκολία του, όσο και για την ικανότητα να παραλληλοποιεί μια εφαρμογή σταδιακά. Περισσότερες λεπτομέρειες θα αναφερθούν στο Κεφάλαιο 2, "Το πρότυπο OpenMP".



Σχήμα 1.2: Μοντέλο Κοινής Μνήμης

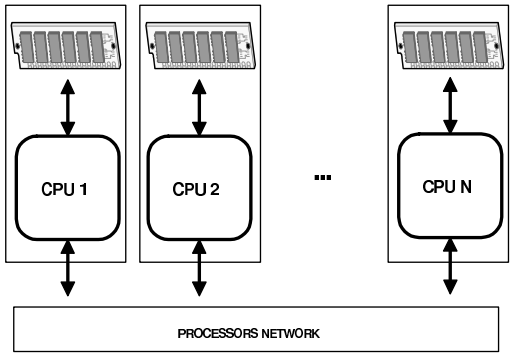
1.2.2 Μοντέλο Μεταβίβασης Μηνυμάτων

Αντίθετα με τους πολυεπεξεργαστές κοινής μνήμης, στα συστήματα κατανεμημένης μνήμης κάθε επεξεργαστής έχει την ιδιωτική του μνήμη την οποία μπορεί να προσπελάσει απευθείας

μόνο αυτός. Η επικοινωνία των επεξεργαστών είναι εφικτή λόγω ύπαρξης διασυνδεδεμένου δικτύου δια μέσω του οποίου γίνεται ανταλλαγή μηνυμάτων. Έτσι, οτιδήποτε χρειαστεί ο επεξεργαστής α από τον επεξεργαστή β (π.χ να προσπελάσει κάποιο δεδομένο από τη μνήμη του β) θα το ζητήσει μέσω μηνύματος από τον β ο οποίος πάλι μέσω μηνύματος που θα ταξιδέψει στο δίκτυο θα το στείλει στον επεξεργαστή α.

Για πολλές εφαρμογές το μοντέλο αυτό είναι αρκετά αποτελεσματικό. Όμως στα προβλήματα που μεγάλος όγκος δεδομένων πρέπει να είναι προσβάσιμος σε πολλές διεργασίες υπάρχει πρόβλημα. Λόγω των πολλών μηνυμάτων που ανταλλάσσονται δημιουργείται μεγάλος φόρτος στο δίκτυο επεξεργαστών με αποτέλεσμα, αντί να υπάρχει βελτίωση στο χρόνο εκτέλεσης σε σύγκριση με το σειριακό μοντέλο, να παρατηρούμε ακόμα πιο αργούς χρόνους. Επιπρόσθετα, ο προγραμματισμός με αυτό το μοντέλο είναι αρκετά πιο περίπλοκος από το μοντέλο κοινής μνήμης.

Δύο από τα πιο δημοφιλή πακέτα που υποστηρίζουν το μοντέλο μεταβίβασης μηνυμάτων είναι το MPI και το PVM. Το πρώτο είναι ένα πρότυπο το οποίο είναι και το πιο διαδεδομένο για προγραμματισμό στο μοντέλο μεταβίβασης μηνυμάτων. Χρησιμοποιεί σταθμούς εργασίας ως επεξεργαστές και μνήμη, και το δίκτυο στο οποίο συνδέονται για επικοινωνία. Διαθέτει έτοιμες ρουτίνες για αποστολή και λήψη μηνυμάτων καθώς και πληθώρα άλλων ευκολιών. Το PVM είναι ένα περιβάλλον το οποίο επιτρέπει παράλληλο δικτυακό προγραμματισμό. Εκτός από ρουτίνες αποστολής και λήψης μηνυμάτων διαθέτει και μηχανισμούς διαχείρισης διεργασιών και λειτουργεί και σε ανομοιογενή περιβάλλοντα. Η χρήση του έχει μειωθεί αισθητά λόγω της επικράτησης του MPI.



Σχήμα 1.3: Μοντέλο Μεταβίβασης Μηνυμάτων

1.3 Αντικείμενο της εργασίας

Το αντικείμενο της εργασίας αφορά την βελτίωση και επέκταση του OMPi [1], ενός μεταγλωττιστή OpenMP για συστήματα UNIX. Καταρχήν το OpenMP είναι μια διεπαφή εφαρμογών προγραμμάτων (API) που παρέχει ένα κλιμακούμενο μοντέλο στους προγραμματιστές κοινής μνήμης. Ο OMPi ως μεταφραστής που υποστηρίζει αυτό το πρότυπο προγραμματισμού χρειάζεται να περιέχει μια γρήγορη βιβλιοθήκη διαχείρισης νημάτων για να εκτελούνται οι παραγόμενες εφαρμογές σε όσο το δυνατό βέλτιστο χρόνο χωρίς επιβάρυνση κατά την εκτέλεση. Με στόχο την απόδοση των προγραμμάτων που παράγει ο μεταγλωττιστής θα τροποποιήσουμε στον κώδικα της βιβλιοθήκης ώστε να κάνει χρήση ατομικών εντολών για να βελτιώσουμε τον χρόνο αυτό.

Οι ατομικές εντολές είναι ένα σύνολο από εντολές γλώσσας μηχανής που αναγνωρίζονται από τον επεξεργαστή σαν μια και μόνο εντολή η οποία τροποποιεί ατομικά μια θέση μνήμης. Με αυτό τον τρόπο κοινές μεταβλητές που προσπελάζονται από πολλά νήματα ταυτόχρονα δεν θα χρειάζονταν να προστατευτούν με αμοιβαίο αποκλεισμό μέσω κλειδαριών και θα τροποποιούνται ατομικά. Έτσι με την αποφυγή κλειδαριών στοχεύουμε σε καλύτερους χρόνους.

Ο OMPi μέχρι στιγμής υποστηρίζει διάφορες βιβλιοθήκες νημάτων όπως Solaris, POSIX, Marcel και Psthreads [2] τόσο για ένα όσο και για πολλά επίπεδα παραλληλίας. Πρόσφατα [7] έχουν γίνει επεκτάσεις στον OMPi προκειμένου να υποστηρίζονται και αρχιτεκτονικές κατανεμημένης μνήμης (π.χ cluster). Σε τέτοια περιβάλλοντα η μονάδα εκτέλεσης δεν είναι πλέον το νήμα αλλά η διεργασία. Είναι επομένως απαραίτητη η δημιουργία μιας βιβλιοθήκης που να υποστηρίζει διαφανώς διεργασίες. Μια τέτοια βιβλιοθήκη είναι το δεύτερο τμήμα της πτυχιακής εργασίας αυτής.

1.4 Δομή της εργασίας

Η εργασία είναι οργανωμένη ως εξής:

- Στο $2^{ον}$ Κεφάλαιο δίνεται μια περιληπτική περιγραφή του προτύπου OpenMP μαζί με τις εντολές που παρέχει.
- Στο $3^{ον}$ Κεφάλαιο παρουσιάζονται οι ατομικές λειτουργίες και οι εντολές που παρέχονται από επεξεργαστές για την υλοποίησή τους. Επίσης δίνονται οι πιθανοί τρόποι ενσωμάτωσης ατομικών εντολών σε εφαρμογές, με τα πλεονεκτήματα και τα μειονεκτήματα τους.
- Στο $4^{ον}$ Κεφάλαιο ακολουθεί μια σύντομη περιγραφή του Ompi compiler. Στο ίδιο κεφάλαιο περιγράφονται οι τροποποιήσεις που πραγματοποιήθηκαν προκειμένου να εισαχθούν ατομικές λειτουργίες.
- Στο $5^{ον}$ Κεφάλαιο κάνουμε μια αποτίμηση του κέρδους που έχουμε από την χρήση ατομικών εντολών. Συγκεκριμένα, παρουσιάζουμε τις εφαρμογές που χρησιμοποιήθηκαν για να πάρουμε συγκριτικές μετρήσεις πριν και μετά την προσθήκη των ατομικών λειτουργιών καθώς και τα αντίστοιχα αποτελέσματα.
- Στο $6^{ον}$ κεφάλαιο γίνεται μια λεπτομερής περιγραφή του τρόπου λειτουργίας του μεταγλωττιστή OMPi μέσω παραδείγματος. Ακολούθως περιγράφεται η υλοποίηση της βιβλιοθήκης διεργασιών.
- Τέλος στο $7^{ον}$ κεφάλαιο υπάρχουν 2 παραρτήματα. Το παράρτημα Α παρουσιάζει μια σύντομη περιγραφή εγκατάστασης του OMPi με επιλογή βιβλιοθήκης. Επιπλέον το παράρτημα Β παρουσιάζει τον όλο κώδικα της βιβλιοθήκης διεργασιών.

Κεφάλαιο 2

Το πρότυπο OpenMP

2.1 Εισαγωγή

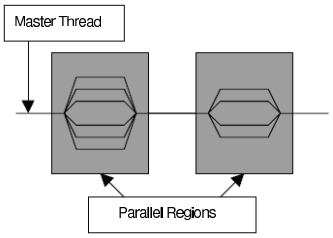
Το OpenMP είναι μια διεπαφή εφαρμογών προγραμμάτων (Application Program Interface – API) το οποίο χρησιμοποιείται για να παραλληλίζουμε προγράμματα σε αρχιτεκτονικές συστημάτων κοινής μνήμης. Αποτελείται από τρία συστατικά: οδηγίες για τον μεταφραστή, συναρτήσεις βιβλιοθηκών και μεταβλητές περιβάλλοντος. Το API του OpenMP έχει οριστεί για τις γλώσσες προγραμματισμού C/C++ και Fortran, ενώ έχει υλοποιηθεί για τις πιο σημαντικές πλατφόρμες όπως το Unix και τα Windows, κάτι που το κάνει αρκετά portable. Έχει οριστεί από κοινού από τις πιο σημαντικές εταιρίες υλικού και λογισμικού καθώς και ακαδημαϊκούς και αναμένεται σε λίγα χρόνια να αποτελέσει πρότυπο ANSI. Το όνομά του προέρχεται από τη φράση Open specifications for Multi Processing που σημαίνει Ανοιχτές προδιαγραφές για Πολύ-Επεξεργασία. Πρέπει να επισημάνουμε ότι το OpenMP δεν προορίζεται για αρχιτεκτονικές συστημάτων κατανεμημένης μνήμης, ούτε έχει υλοποιηθεί από όλες τις εταιρίες υλικού και λογισμικού. Επίσης, δεν έχει σχεδιαστεί για να αποδίδει τη μέγιστη αποδοτικότητα στη χρήση της κοινής μνήμης.

Οι στόχοι του OpenMP είναι να παράσχει ένα πρότυπο που θα ισχύει για αρχιτεκτονικές και πλατφόρμες κοινής μνήμης. Ένα πρότυπο που θα είναι μικρό και εύκολα κατανοητό, δηλαδή θα παρέχει ένα απλό και περιορισμένο σύνολο από οδηγίες με το οποίο θα μπορεί να γίνεται σημαντική παραλληλοποίηση. Επίσης, θα πρέπει να είναι εύκολο στη χρήση του. Το OpenMP παρέχει:
α) τη δυνατότητα παραλληλοποίησης ενός προγράμματος σταδιακά ξεκινώντας από το σειριακό πρόγραμμα και χωρίς να εισάγονται σημαντικές αλλαγές
β) τη δυνατότητα τόσο χονδρού όσο και λεπτού κόκκου παραλληλίας.
Τέλος, ο κυριότερος στόχος του OpenMP είναι η μεταφερσιμότητα (portability).

2.2 Προγραμματιστικό μοντέλο του OpenMP

Το προγραμματιστικό μοντέλο του OpenMP είναι βασισμένο σε παραλληλισμό με νήματα. Μια διεργασία κοινής μνήμης αποτελείται από πολλά νήματα όπου ο προγραμματιστής μπορεί να τα αυξάνει ή να τα μειώνει σταδιακά. Η όλη διαδικασία δε, θυμίζει το μοντέλο fork-join. Όπως φαίνεται στο σχήμα, ένα Master νήμα γεννά ομάδες από νήματα όταν χρειάζεται. Το νήμα master εκτελεί τις εντολές σειριακά μέχρι να φτάσει την πρώτη παράλληλη περιοχή. Η ομάδα των νημάτων που περιλαμβάνεται σε αυτή την παράλληλη περιοχή εκτελείται παράλληλα. Όταν η ομάδα τελειώσει, τότε τα νήματα συγχρονίζονται και τερματίζουν, αφήνοντας μόνο το νήμα

master. Η διαδικασία αυτή επαναλαμβάνεται όσες φορές χρειάζεται.



Σχήμα 2.1: Γενικό σχήμα παραλληλοποίησης στο OpenMP

Τα νήματα επικοινωνούν μεταξύ τους με κοινές μεταβλητές. Ωστόσο, ο διαμοιρασμός των δεδομένων μπορεί να οδηγήσει σε συνθήκες ανταγωνισμού. Για να ελέγξουμε τι συνθήκες ανταγωνισμού, χρησιμοποιούμε συγχρονισμό. Ο συγχρονισμός, όμως, κοστίζει, άρα κάθε φορά που παραλληλοποιούμε πρέπει να οργανώνουμε τα δεδομένα κατά τέτοιο τρόπο ώστε να ελαχιστοποιούμε την ανάγκη για συγχρονισμό.

2.3 Οδηγίες OpenMP για C/C++

Σε αυτό το κεφάλαιο, θα αναφέρουμε τον τρόπο με τον οποίο συντάσσονται οι οδηγίες, θα ασχοληθούμε με κάποιους γενικούς κανόνες πάνω στην χρήση των οδηγιών και με τις παράλληλες περιοχές. Επίσης, θα περιγράψουμε τις συνθήκες (clauses) οδηγιών του OpenMP και θα ασχοληθούμε με τις Περιοχές Διαμοιρασμού Εργασίας (Work-Sharing Constructs). Στο τέλος, θα αναφερθούμε στο συγχρονισμό μεταξύ των δεδομένων.

2.3.1 Σύνταξη Οδηγιών OpenMP

Στον Πίνακα που ακολουθεί φαίνεται η σύνταξη μιας οδηγίας OpenMP.

#pragma omp	**Όνομα Οδηγίας**	**[φρασή 1, ...]**
Απαιτείται για όλες τις οδηγίες C/C++.	Ένα υπαρκτό όνομα οδηγίας. Τοποθετείται μετά το pragma και πριν τις συνθήκες.	Προαιρετικό. Οι συνθήκες μπορεί να εμφανίζονται με κάθε σειρά.

Πίνακας 2.1: Σύνταξη OpenMP οδηγιών

Οι οδηγίες ακολουθούν τις συμβάσεις του προτύπου C/C++. Είναι case sensitive, πρέπει να υπάρχει ένα και μόνο ένα όνομα ανά οδηγία και κάθε οδηγία εφαρμόζεται μόνο στο αμέσως επόμενο block εντολών το οποίο πρέπει να είναι ένα δομημένο block. Επίσης, μακρές οδηγίες μπορούν να συνεχιστούν σε επόμενες γραμμές αρκεί στο τέλος κάθε γραμμής να τοποθετείται ο χαρακτήρας της ανάποδης καθέτου ('\').Προσέξτε ότι οι εντολές #pragma ... αγνοούνται από σειριακούς μεταγλωττιστές (compilers). Με αυτό τον τρόπο, το σειριακό πρόγραμμα ΔΕΝ τροποποιείται.

2.3.2 Παράλληλες Περιοχές

Μια παράλληλη περιοχή είναι ένα τμήμα κώδικα το οποίο θα εκτελεστεί από πολλαπλά νήματα. Η σύνταξη μιας παράλληλης περιοχής είναι η εξής:

```
#pragma omp parallel φράση 1, ...
```

Όταν ένα νήμα φτάσει σε μια οδηγία παραλληλοποίησης, δημιουργεί μια ομάδα από νήματα και το ίδιο γίνεται master της ομάδας. Ο master είναι κι ο ίδιος μέλος της ομάδας. Ο κώδικας της παράλληλης περιοχής εκτελείται αυτούσιος από όλα τα νήματα. Κάθε νήμα μπορεί να φτάσει σε οποιοδήποτε σημείο μέσα στη παράλληλη περιοχή, σε ακαθόριστη χρονική στιγμή. Στο τέλος της παράλληλης περιοχής υπονοείται ένα φράγμα, πέρα από το οποίο μόνο ο master θα συνεχίσει την εκτέλεση του προγράμματος. Εξάλλου, το API παρέχει τη δυνατότητα δημιουργίας παράλληλης περιοχής μέσα σε μια άλλη παράλληλη περιοχή. Η φωλιασμένη αυτή παράλληλη περιοχή, καταλήγει στη δημιουργία μιας καινούριας ομάδας από νήματα η οποία σε πολλές υλοποιήσεις αποτελείται από ένα νήμα. Ωστόσο, διάφορες υλοποιήσεις μπορούν να επιτρέψουν παραπάνω από ένα νήμα σε μια φωλιασμένη παράλληλη περιοχή. Οι υλοποιήσεις αυτές υποστηρίζουν πλήρως τον εμφωλευμένο ή πολυεπίπεδο παραλληλισμό (νεστεδ/μυλτιλεελ παραλλελισμ).

Ο αριθμός των νημάτων που δημιουργούνται κατά την είσοδο σε μια παράλληλη περιοχή μπορεί να καθοριστεί από τρεις παράγοντες, οι οποίοι κατά σειρά προτεραιότητας είναι: α) η χρήση της συνάρτησης βιβλιοθήκης `omp_set_num _threads()`, β) θέτοντας τη μεταβλητή περιβάλλοντος `OMP_NUM_THREADS` και γ) χρησιμοποιώντας την φράση num_threads(). Επίσης, δίνεται η δυνατότητα από το API, να προσαρμόζεται δυναμικά ο αριθμός των νημάτων για μία συγκεκριμένη παράλληλη περιοχή. Αυτό μπορεί να επιτευχθεί με τους εξής δύο τρόπους: α) με τη χρήση της συνάρτησης βιβλιοθήκης `omp_set_dynamic()` και β) θέτοντας τη μεταβλητή περιβάλλοντος `OMP_DYNAMIC`.

2.3.3 Συνθήκες Οδηγιών του OpenMP

Οι κυριότερες φράσεις οδηγιών που χρησιμοποιούνται στο OpenMP είναι οι εξής:

- **shared(var1,var2,...):** Κοινόχρηστες μεταβλητές (var1,var2,...) οι οποίες θα προσπελαθούν από όλα τα νήματα (τα νήματα προσπελαύνουν ίδιες θέσεις μνήμης).
- **private(var1,var2,...):** Κάθε νήμα έχει το δικό του αντίγραφο από αυτές τις ιδιωτικές μεταβλητές (var1,var2,...), για τη διάρκεια της εκτέλεσης της παράλληλης περιοχής.
- **firstprivate(var1,var2,...):** Ιδιωτικές μεταβλητές (var1,var2,...) που αρχικοποιούνται όταν εισάγεται μια παράλληλη περιοχή.
- **if(expression):** Η παραλληλοποίηση γίνεται μόνο όταν το expression είναι αληθές.
- **schedule(type [,chunk]):** Ελέγχει τον τρόπο με τον οποίο οι επαναλήψεις ενός βρόγχου θα διαμοιραστούν στα νήματα. Το type του schedule μπορεί να είναι:
  - **STATIC:** Οι επαναλήψεις ενός βρόγχου διαμοιράζονται σε κομμάτια μεγέθους chunk και ανατίθεται στατικά στα νήματα. Αν το chunk δεν έχει καθοριστεί, τότε οι επαναλήψεις διαμοιράζονται ισομερώς (αν αυτό είναι δυνατό) και συνεχόμενα μεταξύ των νημάτων.

- **DYNAMIC:** Οι επαναλήψεις ενός βρόγχου διαμοιράζονται σε κομμάτια μεγέθους chunk και ανατίθεται δυναμικά στα νήματα. Όταν ένα νήμα τελειώσει με ένα κομμάτι αναλαμβάνει δυναμικά ένα άλλο. Το default μέγεθος ενός κομματιού είναι 1 επανάληψη.
- **GUIDED:** Το μέγεθος του κομματιού μειώνεται δυναμικά με κάθε κομμάτι του χώρου επαναλήψεων που αποστέλλεται στα νήματα. Το μέγεθος chunk καθορίζει τον ελάχιστο αριθμό επαναλήψεων που θα πρέπει να αποσταλούν κάθε φορά. Το default μέγεθος ενός κομματιού είναι 1.
- **RUNTIME:** Η απόφαση διαμοιρασμού, αναβάλλεται μέχρι την εκτέλεση του προγράμματος με τη βοήθεια της μεταβλητής περιβάλλοντος OMP_SCHEDULE. Απαγορεύεται να καθοριστεί μέγεθος chunk για αυτή τη συνθήκη.

- **reduction(operator—intrinsic:var1,var2,...):** Εξασφαλίζει ότι μια reduction λειτουργία (του operator) μεταξύ των μεταβλητών var1, var2, ... θα εκτελεστεί με ασφάλεια (για παράδειγμα ένα καθολικό άθροισμα).

2.3.4 Περιοχές Διαμοιρασμού Εργασίας

Μια περιοχή διαμοιρασμού εργασίας χωρίζει τον κώδικα που περικλείεται στην παράλληλη περιοχή μεταξύ των νημάτων της ομάδας της περιοχής. Οι περιοχές διαμοιρασμού εργασίας δεν δημιουργούν καινούρια νήματα και δεν υπάρχει κάποιος συγχρονισμός κατά την είσοδο σε μια τέτοια περιοχή, υπάρχει όμως συγχρονισμός κατά την έξοδο. Υπάρχουν τρεις τύποι περιοχών διαμοιρασμού εργασίας:

- **Οδηγία for:** Διαμοιράζει τις επαναλήψεις ενός βρόγχου στην ομάδα της τρέχουσας παράλληλης περιοχής. Αντιπροσωπεύει τον τύπο "Παραλληλισμού Δεδομένων".
- **Οδηγία sections:** "Σπάει" την εργασία σε ξεχωριστά, διακριτά τμήματα. Κάθε τμήμα εκτελείται από διαφορετικό νήμα. Μπορεί να χρησιμοποιηθεί για να υλοποιηθεί ο τύπος του "Συναρτησιακού Παραλληλισμού".
- **Οδηγία single:** Ένα τμήμα του κώδικα που υπάρχει σε μια παράλληλη περιοχή μπορεί να εκτελεστεί μόνο από ένα νήμα.

Όπως και στις παράλληλες περιοχές, έτσι και στις περιοχές διαμοιρασμού εργασίας, υπάρχουν κάποιοι περιορισμοί που πρέπει να ισχύουν. Πρώτον, για να εκτελεσθεί παράλληλα μια οδηγία, θα πρέπει η περιοχή διαμοιρασμού εργασίας να εσωκλείεται δυναμικά μέσα σε μια παράλληλη περιοχή. Δεύτερον, οι περιοχές διαμοιρασμού εργασίας θα πρέπει να διαμοιράζουν τα δεδομένα (ή την εργασία) σε όλα τα νήματα ή σε κανένα. Τρίτον, διαδοχικές περιοχές διαμοιρασμού εργασίας θα πρέπει να προσπελαύνονται από τα μέλη μιας ομάδας με την ίδια σειρά.

2.3.5 Οδηγίες Συγχρονισμού για C/C++

Οι Οδηγίες Συγχρονισμού χρησιμοποιούνται για να ελέγξουμε τη σειρά εκτέλεσης των νημάτων μιας ομάδας, έτσι ώστε να εξασφαλίσουμε την ακεραιότητα των δεδομένων. Οι οδηγίες συγχρονισμού που μας δίνει το API είναι οι εξής:

- **master** οδηγία `#pragma omp master`
  Η οδηγία master καθορίζει μια περιοχή η οποία θα εκτελεστεί μόνο από το νήμα master της ομάδας. Όλα τα άλλα νήματα της ομάδας δεν εκτελούν αυτό το τμήμα κώδικα. Για την οδηγία αυτή, δεν υπάρχει κάποιο φράγμα που να υπονοείται.

- **critical** οδηγία `#pragma omp critical [name]`
  Η οδηγία critical καθορίζει μια περιοχή η οποία πρέπει να εκτελεστεί μόνο από ένα νήμα κάθε φορά. Χρησιμοποιείται για να ορίσουμε μια κρίσιμη περιοχή. Σε μια κρίσιμη περιοχή, μόνο μια διεργασία μπορεί να εγγράψει ή να διαβάσει μια κοινή μεταβλητή διασφαλίζοντας έτσι την ακεραιότητα αυτής της μεταβλητής.

- **barrier** οδηγία `#pragma omp barrier`
  Η οδηγία barrier συγχρονίζει όλα τα νήματα της ομάδας απαιτώντας από κάθε νήμα να σταματήσει, προσωρινά την εκτέλεσή του, στο σημείο όπου υπάρχει η barrier οδηγία, μέχρις ότου όλα τα νήματα φτάσουν σε αυτό το σημείο. Στη συνέχεια, όλα τα νήματα ξεκινούν παράλληλα, από εκείνο το σημείο, την εκτέλεσή του κώδικα που ακολουθεί.

- **atomic** οδηγία `#pragma omp atomic`
  Η atomic οδηγία καθορίζει ότι η συγκεκριμένη τοποθεσία στη μνήμη πρέπει να ανανεώνεται ατομικά (από κάθε νήμα), μην επιτρέποντας πολλά νήματα να κάνουν εγγραφή στη συγκεκριμένη τοποθεσία. Έτσι απαγορεύει σε οποιοδήποτε νήμα να διακόψει κάποιο άλλο νήμα που βρίσκεται στη διαδικασία προσπέλασης ή αλλαγής της τιμής μιας μεταβλητής κοινής μνήμης.

- **flush** οδηγία `#pragma omp flush`
  Η flush οδηγία καθορίζει ένα σημείο συγχρονισμού στο οποίο η υλοποίηση του κώδικα πρέπει να παρέχει ένα συνεπές στιγμιότυπο της μνήμης. Σε αυτό το σημείο η τρέχουσα τιμή μιας κοινής μεταβλητής εγγράφεται αμέσως στη μνήμη (write back).

- **ordered** οδηγία `#pragma omp ordered`
  Η οδηγία ordered καθορίζει ότι οι επαναλήψεις του εσωκλειώμενου βρόγχου θα εκτελεστούν με τη σειρά όπως θα εκτελούνταν σε ένα σειριακό υπολογιστή.

2.3.6 Εξειδικευμένη Οδηγία: threadprivate

Η οδηγία threadprivate καθορίζει ότι καθολικά αντικείμενα (ή μεταβλητές) μπορούν να γίνουν ιδιωτικά για κάποιο νήμα. Με αυτό τον τρόπο, μπορούμε να ορίσουμε καθολικά αντικείμενα, και να μετατρέψουμε την εμβέλειά τους σε τοπική. Οι μεταβλητές για τις οποίες ισχύει η οδηγία threadprivate συνεχίζουν να είναι ιδιωτικές, για κάθε νήμα, ακόμα και σε διαφορετικές παράλληλες περιοχές. Η σύνταξη της threadprivate οδηγίας είναι η παρακάτω:

```
#pragma omp threadprivate
```

Η οδηγία πρέπει να εμφανίζεται αμέσως μετά τις δηλώσεις των καθολικών μεταβλητών. Μετά την οδηγία, η συγκεκριμένη μεταβλητή που έχει γίνει threadprivate, αντιγράφεται σε κάθε νήμα και κάθε νήμα κρατάει το δικό του αντίγραφο, έτσι ώστε δεδομένα τα οποία εγγράφονται από ένα νήμα να μην είναι ορατά στα υπόλοιπα.

2.4 Συναρτήσεις Βιβλιοθήκης Runtime

Το OpenMP παρέχει κάποιες συναρτήσεις βιβλιοθήκης υποστήριξης εκτέλεσης (runtime). Οι συναρτήσεις μεταξύ άλλων παρέχουν το κλείδωμα κρίσιμων περιοχών, τον αριθμό των τρεχόντων νημάτων και τη δυναμική προσαρμογή τους. Ειδικότερα, για τις συναρτήσεις κλειδώματος, η μεταβλητή (κλειδαριά) μπορεί να προσπελαστεί μόνο με τις συναρτήσεις κλειδώματος που παρέχονται από τη βιβλιοθήκη. Επίσης, η μεταβλητή, θα πρέπει αν είναι συγκεκριμένου τύπου, `omp_lock_t` ή `omp_nest_lock_t`. Υπάρχουν συναρτήσεις κλειδαριών τόσο για απλά όσο και για εμφωλευμένα επίπεδα. Οι κυριώτερες συναρτήσεις βιβλιοθήκης Runtime είναι:

- `void omp_set_num_threads (int num_threads)`

Θέτει τον αριθμό των νημάτων που θα χρησιμοποιηθούν στην επόμενη παράλληλη περιοχή. Παίρνει σαν όρισμα έναν ακέραιο αριθμό ο οποίος εκφράζει τον αριθμό των νημάτων. Ο δυναμικός μηχανισμός των νημάτων μπορεί να τροποποιήσει το αποτέλεσμα της συνάρτησης αυτής. Έτσι, αν ο μηχανισμός αυτός είναι "Enabled", τότε η συνάρτηση καθορίζει το μέγιστο αριθμό νημάτων που μπορεί αν χρησιμοποιηθεί από κάθε περιοχή. Αν ο μηχανισμός είναι "Disabled", τότε η συνάρτηση καθορίζει τον ακριβή αριθμό νημάτων προς χρήση, μέχρι την επόμενη κλήση της συνάρτησης. Η συνάρτηση αυτή, μπορεί να κληθεί μόνο από κάποιο σειριακό κομμάτι κώδικα και έχει προτεραιότητα έναντι της μεταβλητής περιβάλλοντος `OMP_NUM_THREADS` (βλέπε παράγραφο 2.5).

- `int omp_get_num_threads (void)`

Επιστρέφει τον αριθμό (ακέραιος) των νημάτων που βρίσκονται τη δεδομένη στιγμή στην ομάδα που εκτελεί την τρέχουσα παράλληλη περιοχή. Αν αυτή η συνάρτηση εκτελεστεί από σειριακό κομμάτι του κώδικα ή από φωλιασμένη παράλληλη περιοχή, τότε θα επιστρέψει τιμή 1.

- `void omp_get_thread_num(void)`

Επιστρέφει τον αριθμό εκείνου του νήματος, μέσα σε μια ομάδα, το οποίο έκανε την κλήση της συνάρτησης. Η αρίθμηση των νημάτων ξεκινάει από το 0 (που το έχει το νήμα master) εως το `OMP_NUM_THREADS - 1`. Η συνάρτηση θα επιστρέψει 0 αν κληθεί μέσα από ένα σειριακό κομμάτι κώδικα.

- `void omp_set_dynamic(int dynamic_threads)`

Επιτρέπει ή απαγορεύει την δυναμική ρύθμιση του αριθμού των νημάτων (από το runtime σύστημα) που είναι διαθέσιμα για εκτέλεση από τις παράλληλες περιοχές. Παίρνει σαν όρισμα έναν ακέραιο αριθμό. Αν ο αριθμός είναι διάφορος του 0, τότε ο μηχανισμός δυναμικής ρύθμισης του αριθμού των νημάτων, από το runtime σύστημα, ενεργοποιείται, αλλιώς, αν είναι 0, απενεργοποιείται. Η αρχική κατάσταση του μηχανισμού αυτού εξαρτάται από την υλοποίηση. Η συνάρτηση μπορεί να κληθεί από κάποιο σειριακό κομμάτι του κώδικα και έχει προτεραιότητα έναντι της μεταβλητής περιβάλλοντος `OMP_DYNAMIC` (βλέπε παράγραφο 2.5).

- `int omp_get_dynamic(void)`

Επιστρέφει 1 αν είναι ενεργοποιημένος ο μηχανισμός δυναμικής ρύθμισης του αριθμού των νημάτων από το runtime σύστημα και 0 αν είναι απενεργοποιημένος.

- `void omp_set_nested(int nested)`

Ενεργοποιεί ή απενεργοποιεί τη δυνατότητα παράλληλου φωλιάσματος. Παίρνει σαν όρισμα ένα ακέραιο αριθμό. Αν ο αριθμός είναι διάφορος του 0, τότε η δυνατότητα παράλληλου φωλιάσματος ενεργοποιείται, αλλιώς, αν η τιμή είναι 0, απενεργοποιείται. Η συνάρτηση έχει προτεραιότητα έναντι της μεταβλητής περιβάλλοντος `OMP_NESTED` (βλέπε παράγραφο 2.5).

- `int omp_get_nested(void)`

Επιστρέφει 1 αν είναι ενεργοποιημένη η δυνατότητα παράλληλου φωλιάσματος και 0 αν είναι απενεργοποιημένη.

- `void omp_init_lock (omp_lock_t *lock)`

Αρχικοποιεί μια κλειδαριά στην οποία αναφέρεται ο δείκτης lock. Η αντίστοιχη συνάρτηση για φωλιασμένη κλειδαριά είναι `void omp_nest_init_lock (omp_nest_lock_t *lock)`. Η αρχική κατάσταση της κλειδαριάς είναι "unlock".

- `void omp_set_lock (omp_lock_t *lock)`

Αναγκάζει το νήμα που εκτελείται να περιμένει μέχρις ότου η κλειδαριά στην οποία δείχνει το lock, να γίνει διαθέσιμη. Η αντίστοιχη συνάρτηση για φωλιασμένη κλειδαριά είναι `void omp_nest_set_lock (omp_nest_lock_t *lock)`. Απαγορεύεται να κληθεί αυτή η συνάρτηση με μεταβλητή lock η οποία δεν έχει αρχικοποιηθεί.

- `void omp_unset_lock (omp_lock_t *lock)`

Αποδεσμεύει τη κλειδαριά από το νήμα που την χρησιμοποιούσε. Η αντίστοιχη συνάρτηση για φωλιασμένη κλειδαριά είναι `void omp_nest_unset_lock (omp_nest_lock_t *lock)`. Απαγορεύεται να κληθεί αυτή η συνάρτηση με μεταβλητή lock η οποία δεν έχει αρχικοποιηθεί.

- `int omp_test_lock (omp_lock_t *lock)`

Προσπαθεί να αρχικοποιήσει μια κλειδαριά, αλλά δεν κάνει block το νήμα που εκτελεί αυτή τη συνάρτηση, ακόμα κι αν η κλειδαριά δεν είναι διαθέσιμη. Επιστρέφει 1 αν η κλειδαριά έχει αρχικοποιηθεί επιτυχώς και 0 σε αντίθετη περίπτωση. Η αντίστοιχη συνάρτηση για φωλιασμένη κλειδαριά είναι `void omp_nest_test_lock (omp_nest_lock_t *lock)`. Απαγορεύεται να κληθεί αυτή η συνάρτηση με μεταβλητή lock η οποία δεν έχει αρχικοποιηθεί.

2.5 Μεταβλητές Περιβάλλοντος

Το OpenMP παρέχει τέσσερις μεταβλητές περιβάλλοντος που μας βοηθάνε στον έλεγχο της εκτέλεσης του παράλληλου κώδικα. Όλες οι μεταβλητές περιβάλλοντος γράφονται με κεφαλαία γράμματα και οι τιμές που τους ανατίθενται δεν είναι "case sensitive". Οι τέσσερις αυτές μεταβλητές περιβάλλοντος είναι:

- `OMP_SCHEDULE`

  Η τιμή αυτής της μεταβλητής καθορίζει τον τρόπο με τον οποίο οι επαναλήψεις ενός βρόγχου κατανέμονται στους επεξεργαστές. Ισχύει μόνο για τις οδηγίες `for` και `parallel for` στις οποίες η schedule συνθήκη έχει τεθεί στο "runtime".

- `OMP_NUM_THREADS`

  Η τιμή αυτής της μεταβλητής θέτει τον μέγιστο αριθμό νημάτων που μπορεί να εκτελέσει μια παράλληλη περιοχή.

- `OMP_DYNAMIC`

  Η τιμή αυτής της μεταβλητής ενεργοποιεί ή απενεργοποιεί τη δυναμική ρύθμιση του αριθμού των νημάτων (από το runtime σύστημα), που είναι διαθέσιμα προς εκτέλεση από τις παράλληλες περιοχές. Οι έγκυρες τιμές είναι true, αν θέλουμε ο μηχανισμός αυτός να ενεργοποιηθεί και false αν θέλουμε να απενεργοποιηθεί.

- `OMP_NESTED`

  Η τιμή αυτής της μεταβλητής ενεργοποιεί ή απενεργοποιεί τη δυνατότητα παράλληλου φωλιάσματος. Οι έγκυρες τιμές είναι true, αν θέλουμε η δυνατότητα παράλληλου φωλιάσματος να ενεργοποιηθεί και false αν θέλουμε να απενεργοποιηθεί. Σε υλοποιήσεις που δεν υποστηρίζουν εμφωλευμένο παραλληλισμό, η μεταβλητή αυτή δεν παίζει κανένα απολύτως ρόλο.

Κεφάλαιο 3

Ατομικές Λειτουργίες

3.1 Ορισμός

Οι ατομικές λειτουργίες είναι ένα σύνολο από εντολές που αναγνωρίζονται από τον επεξεργαστή ως μια και αδιάσπαστη εντολή, η οποία τροποποιεί ατομικά μια θέση μνήμης. Δηλαδή καμία διεργασία δεν μπορεί να προσπελάσει με οποιοδήποτε τρόπο τη συγκεκριμένη θέση μνήμης μέχρι την ολοκλήρωση της ατομικής εντολής.

Για να επιτευχθεί αυτός ο μηχανισμός θα πρέπει να ισχύουν τα ακόλουθα:

- Πριν ολοκληρωθεί το σύνολο των εντολών καμιά άλλη διεργασία δεν μπορεί να ξέρει τις αλλαγές που έχουν γίνει.
- Αν μια από τις εντολές αποτύχει τότε όλες οι εντολές αποτυγχάνουν και το σύστημα θα πρέπει να επανέλθει στην κατάσταση που βρισκόταν προτού εκτελεστεί η ατομική εντολή.

Οι ατομικές εντολές, όπως θα αναφερθεί και πιο κάτω, χρησιμοποιούνται σε παράλληλα συστήματα κοινής μνήμης για τροποποίηση της τιμής μιας μεταβλητής χωρίς τη χρήση κλειδαριών ή σημαφόρων. Δηλαδή διευκολύνουν την ανάπτυξη wait-free και lock-free αλγορίθμων, επιτρέποντας την αδιάκοπη λειτουργία μιας διεργασίας ή ενός νήματος και μειώνοντας έτσι τις καθυστερήσεις λόγω του ανταγωνισμού των νημάτων. Η απουσία αυτή των κλειδαριών έχει ως αποτέλεσμα την αποδοτικότερη εκτέλεση παράλληλων εφαρμογών, ειδικότερα στην περίπτωση που πολλές διεργασίες ή νήματα, προσπαθούν να τροποποιήσουν ταυτόχρονα μια κοινή μεταβλητή.

3.2 Συστήματα κοινής μνήμης & Κοινές Μεταβλητές

Η αρχιτεκτονική των συστημάτων κοινής μνήμης ή κοινού χώρου διευθύνσεων βασίζεται σε ένα ενιαίο χώρο μνήμης στον οποίο έχει πρόσβαση ένας αριθμός επεξεργαστών. Ένα από τα προβλήματα που αντιμετωπίζουμε σε αυτά τα συστήματα είναι η προσπέλαση μιας κοινής μεταβλητής.

Σε συστήματα ενός επεξεργαστή, η μνήμη προσπελαύνεται σειριακά με αποτέλεσμα να μην υφίστανται προβλήματα κατά την τροποποίησή της. Ωστόσο σε συστήματα κοινής μνήμης με πολλούς επεξεργαστές είναι αδύνατο να προβλέψουμε τη συμπεριφορά μιας παράλληλης

εφαρμογής που προσπελαύνει μέσο πολλαπλών νημάτων ταυτόχρονα μια θέση μνήμης. Ένα απλό παράδειγμα τέτοιου προβλήματος παρουσιάζεται και παρακάτω.

Έστω δυο διεργασίες που διαμοιράζονται ένα χώρο κοινής μνήμης προσπαθούν να αυξήσουν κατά 1 την τιμή μιας μεταβλητής.

A1. Η πρώτη διεργασία διαβάζει την τιμή της μεταβλητής από τη διεύθυνσή μνήμης.

A2. Η πρώτη διεργασία αυξάνει τη τιμή κατά 1.

Προτού όμως μπορέσει να αποθηκεύσει την αλλαγή, η διεργασία τίθεται σε αναμονή από το λειτουργικό σύστημα και επιτρέπεται στη δεύτερη διεργασία να εκτελεστεί:

B1. Η δεύτερη διεργασία διαβάζει την τιμή της μεταβλητής από τη διεύθυνσή μνήμης (ίδια με τη τιμή που διάβασε και η πρώτη διεργασία).

B2. Η δεύτερη διεργασία αυξάνει τη τιμή κατά 1.

B3. Η δεύτερη διεργασία αποθηκεύει την τιμή στη μνήμη

Τώρα η πρώτη διεργασία ενεργοποιείται και της επιτρέπεται να συνεχίσει την εκτέλεσή της:

A3. Η πρώτη διεργασία αποθηκεύει μια λάθος τιμή στη μνήμη, χωρίς να γνωρίζει ότι η μεταβλητή έχει ήδη αλλάξει.

Αυτό είναι ένα τετριμμένο παράδειγμα που σε πραγματικές συνθήκες μπορεί να γίνει ακόμα πιο περίπλοκο. Τέτοια προβλήματα μπορούμε εύκολα να τα αποφύγουμε με τη χρήση κλειδαριών.

3.3 Ατομικές εντολές επεξεργαστών

Παραδείγματα ατομικών εντολών υπάρχουν σε όλους τους σύγχρονους επεξεργαστές, με διαφορετική υλοποίηση στον κάθε ένα. Για παράδειγμα οι επεξεργαστές της Intel [12] (Intel x86) και AMD υποστηρίζουν την εντολή LOCK η οποία μετατρέπει την εντολή που ακολουθεί σε ατομική ενεργοποιώντας το σήμα διαύλου LOCK#. Με τον τρόπο αυτό, σε ένα πολυεπεξεργαστικό περιβάλλον εξασφαλίζεται η αποκλειστική χρήση του διαύλου (και άρα της κοινής μνήμης) από ένα μονό επεξεργαστή κατά την εκτέλεση της επερχόμενης εντολής.

Οι εντολές οι οποίες μπορούν να γίνουν ατομικά με το σήμα LOCK# είναι οι ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD και XCHG. Από αυτές ιδιαίτερο ενδιαφέρον παρουσιάζουν οι εντολές CMPXCHG και XCHG με τις οποίες υλοποιούνται οι κλειδαριές στα περισσότερα συστήματα.

Η εντολή XCHG έχει την ιδιαιτερότητα ότι πάντοτε ενεργοποιεί το σήμα LOCK# ανεξάρτητα αν αυτό έχει ενεργοποιηθεί η όχι, εφόσον ο προορισμός και η πηγή είναι θέσεις μνήμης και όχι καταχωρητές.

Ο ψευδοκώδικας που ακολουθεί παρουσιάζει την υλοποίηση των εντολών XCHG και CMPXCHG.

- XCHG—Exchange Register/Memory with Register(Intel x86)
  Ανταλλαγή καταχωρητή/θέσης μνήμης με καταχωρητή

  Σύνταξη εντολής: `XCHG DEST, SRC`

  Αλγόριθμος εκτέλεσης:
  `TEMP ← DEST  /* TEMP = temporary variable */`

  `DEST ← SRC`

  `SRC ← TEMP`

  Ο προορισμός και η πηγή μπορούν να είναι είτε καταχωρητής, είτε θέση μνήμης.

- CMPXCHG—Compare and Exchange (Intel x86)
  Σύγκριση και ανταλλαγή

  Σύνταξη εντολής: `CMPXCHG DEST, SRC`

  Αλγόριθμος εκτέλεσης:
  `IF accumulator = DEST  /*accumulator = AL, AX, or EAX*/`

  `THEN`

  `ZF ← 1  /* ZP flag */`

  `DEST ← SRC`

  `ELSE`

  `ZF ← 0`

  `accumulator ← DEST`

  FI;

Μια πιο λιτή υποστήριξη ατομικών εντολών παρέχεται από την Sun. Οι επεξεργαστές της Sun (SparcV9) παρέχουν τρεις εντολές οι οποίες εγγυώνται πάντοτε την ατομική εκτέλεσή τους.

- Compare and Swap (CASA, CASXA)
  Ίδια λειτουργία με την CMPXCHG της Intel.

- Load Store Unsigned Byte (LDSTUB, LDSTUBA)
  Η LDSTUB αντιγράφει τα περιεχόμενα μιας θέσης μνήμης σε ένα καταχωρητή και γράφει στη μνήμη την τιμή 0xFF (hexadecimal).

- Swap (SWAP, SWAPA)
  Ίδια λειτουργία με την XCHG της Intel.

3.4 Χρήση ατομικών λειτουργιών και κώδικα C

Οι ατομικές εντολές πρέπει με κάποιο τρόπο να ενσωματώνονται μέσα στον κώδικα μιας εφαρμογής. Οι διαθέσιμοι τρόποι που μπορεί να γίνει αυτό είναι τρεις. Όλοι παρουσιάζουν πλεονεκτήματα και μειονεκτήματα και αναλύονται πιο κάτω.

3.4.1 Κώδικα μηχανής – Assembly

Είναι προφανές ότι οι ατομικές εντολές υλοποιούνται σε κώδικα μηχανής χαμηλού επιπέδου και είναι διαφορετικές για κάθε επεξεργαστή. Αυτό έχει ως αποτέλεσμα οι παράλληλες εφαρμογές που υλοποιούνται με ατομικές εντολές επεξεργαστή να μην είναι φορητές και να παρουσιάζουν ιδιαίτερη δυσκολία στον προγραμματιστή αφού θα πρέπει να συντάξει κώδικα μηχανής, πράγμα εξαιρετικά δύσκολο. Γι΄ αυτό το λόγο έχουν δημιουργηθεί έμμεσοι τρόποι εκτέλεσης ατομικών εντολών επεξεργαστή από υψηλό επίπεδο προγραμματισμού και ανεξάρτητοι από τον τύπο του επεξεργαστή. Όμως το θετικό αυτού του είδους προγραμματισμού είναι η δημιουργία βέλτιστων αλγορίθμων. Ο προγραμματιστής γνωρίζει την υλοποίηση του αλγορίθμου του και μπορεί να τον προσαρμόσει ακριβώς στις ανάγκες της εφαρμογής, πάντα σε συγκεκριμένο μηχάνημα.

3.4.2 Έτοιμες βιβλιοθήκες

Ένας ενδιαφέρων τρόπος είναι μέσω βιβλιοθηκών που παρέχουν έτοιμες συναρτήσεις υλοποίησης ατομικών εντολών. Κάτω από αυτές τις συναρτήσεις κρύβεται κώδικας που αναγνωρίζει τον τύπο του επεξεργαστή και την υποστήριξή του για ατομικές εντολές και εκτελεί τον ανάλογο κώδικα μηχανής σε κάθε περίπτωση. Αν δεν υπάρχει υποστήριξη της συγκεκριμένης ατομικής εντολής τότε υλοποιείται έμμεσα με τις υπάρχουσες εντολές του συγκεκριμένου επεξεργαστή (π.χ η SWAP μπορεί να γίνει με την XCHG και αντίθετα). Σε οποιαδήποτε άλλη περίπτωση εκτελείται ο κώδικας με απλές κλειδαριές (locks) επιτυγχάνοντας έτσι αμοιβαίο αποκλεισμό, και προστατεύοντας την κρίσιμη περιοχή, με μειωμένη πάντως απόδοση.

Μια τέτοια βιβλιοθήκη είναι η Netscape Portable Runtime (NSPR) από τους Wan-Teh Chang και Darin Fisher[9]. Η NSPR είναι μια διεπαφή εφαρμογών προγραμμάτων (API) σε γλώσσα C που χρησιμοποιείται από τον Mozila, το Netscape και άλλες εφαρμογές και παρέχει μεταξύ άλλων τις ακόλουθες ατομικές εντολές:

- `PRInt32 PR_AtomicIncrement(PRInt32 *val);`
  Ατομική αύξηση κατά 1 της τιμής στη θέσης μνήμης που καθορίζεται από τον δείκτη val και επιστροφή της νέας τιμής.

- `PRInt32 PR_AtomicDecrement(PRInt32 *val);`
  Ατομική μείωση κατά 1 της τιμής στη θέσης μνήμης που καθορίζεται από τον δείκτη val και επιστροφή της νέας τιμής.

- `PRInt32 PR_AtomicSet(PRInt32 *val,PRInt32 newval);`
  Ατομική εκχώρηση στη θέση μνήμης που καθορίζεται από το δείκτη val της newval και επιστροφή της τιμής που βρίσκεται στη μνήμη πριν από την αλλαγή.

Μια ακόμα βιβλιοθήκη που περιέχει ατομικές εντολές είναι η GLib της Gnome από τους Owen Taylor και Tim Janik[10]. Και αυτή η βιβλιοθήκη είναι υλοποιημένη στη γλώσσα C και παρέχει πληθώρα χρήσιμων συναρτήσεων, με τις ακόλουθες ατομικές συναρτήσεις.

- `gint g_atomic_int_get(volatile gint *atomic);`
  Επιστροφή της τιμής του ακεραίου που δείχνει ο δείκτης `atomic`.

- `void g_atomic_int_set(volatile gint *atomic, gint newval);`
  Ατομική εκχώρηση του ακεραίου `newval` στη θέση μνήμης που δείχνει ο δείκτης `atomic`.

- `void g_atomic_int_add(volatile gint *atomic, gint val);`
  Ατομική πρόσθεση του ακεραίου `newval` στη τιμή της θέσης μνήμης που δείχνει ο δείκτης `atomic`. Επιστροφή της τιμής στη θέση μνήμης πριν γίνει η αλλαγή.

- `gint g_atomic_int_exchange_and_add(volatile gint *atomic, gint oldval, gint newval);`
  Ατομική πρόσθεση του ακεραίου `newval` στη τιμή της θέσης μνήμης που δείχνει ο δείκτης `atomic`. Επιστροφή της τιμής στη θέση μνήμης πριν γίνει η αλλαγή.

- `gboolean g_atomic_int_compare_and_exchange(volatile gint *atomic, gint oldval, gint newval);`
  Σύγκριση της τιμής στη θέση μνήμης που δείχνει ο δείκτης `atomic` με την τιμή `oldval`, και αν είναι ίσες πραγματοποιείται ατομική εναλλαγή του `*atomic` με `newval`. Επιστροφή `true` αν ο `oldval` είναι ίσος με `*atomic`.

- `gpointer g_atomic_pointer_get(volatile gpointer *atomic);`
  Επιστροφή της τιμής στη θέση μνήμης που δείχνει ο δείκτης `*atomic`.

- `void g_atomic_pointer_set(volatile gpointer *atomic, gpointer newval);`
  Αλλαγή της τιμής του δείκτη που δείχνει ο `atomic` σε `newval`.

- `gboolean g_atomic_pointer_compare_and_exchange(volatile gpointer *atomic, gpointer oldval,gpointer newval);`
  Σύγκριση του δείκτη που δείχνει ο δείκτης `atomic` με τον δείκτη `oldval`, και αν είναι ίσοι πραγματοποιείται ατομική εναλλαγή του `*atomic` σε `newval`. Επιστροφή `true` αν ο `oldval` είναι ίσος με `*atomic`.

- `void g_atomic_int_inc(gint *atomic);`
  Ατομική αύξηση της θέσης μνήμης που δείχνει ο δείκτης `atomic` κατά 1.

- `gboolean g_atomic_int_dec_and_test(gint *atomic);`
  Ατομική μείωση της θέσης μνήμης που δείχνει ο δείκτης `atomic` κατά 1 και επιστροφή `true` αν ο ακέραιος στη μνήμη γίνει 0 μετά τη μείωση.

Επίσης μια τρίτη βιβλιοθήκη που παρέχει ατομικές εντολές είναι το *The atomic_ops project* [11] από την HP. Αυτή η βιβλιοθήκη σε σχέση με τις προηγούμενες αποτελείται μόνο από ατομικές εντολές. Οι ατομικές εντολές που περιέχει είναι οι ακόλουθες:

- `void nop()`
  Δεν παρέχει καμία ατομική εντολή. Αποτελεί μια barrier εντολή.

- `AO_t load(volatile AO_t * addr)`
  Διαβάζεται ατομικά η μεταβλητή στην διεύθυνση μνήμης που δείχνει ο δείκτης `addr`.

- `void store(volatile AO_t * addr, AO_t new_val)`
  Ατομική αποθήκευση της τιμής `new_val` στη διεύθυνση μνήμης που δείχνει ο δείκτης `addr`.

- `AO_t fetch_and_add(volatile AO_t *addr, AO_t incr)`
  Ατομική αύξηση της τιμής στη θέση μνήμης που δείχνει ο δείκτης `addr` κατά την τιμή της `new_val`. Επιστροφή της τιμής πριν την αλλαγή στη θέση μνήμης που δείχνει ο δείκτης `addr`.

- `AO_t fetch_and_add1(volatile AO_t *addr)`
  Ισοδύναμο με την εντολή `AO_fetch_and_add(addr, 1)`.

- `AO_t fetch_and_sub1(volatile AO_t *addr)`
  Ισοδύναμο με την εντολή `AO_fetch_and_add(addr, (AO_t)(-1))`.

- `void or(volatile AO_t *addr, AO_t incr)`
  Ατομική πράξη OR της τιμής `incr` στη τιμή που δείχνει ο δείκτης `addr`.

- `int compare_and_swap(volatile AO_t * addr, AO_t old_val, AO_t new_val)`
  Ατομική σύγκριση τις τιμής που δείχνει ο δείκτης `addr` με την τιμή `old_val`. Αν η σύγκριση είναι επιτυχής αντικαθιστά την τιμή που δείχνει ο δείκτης `addr` με την τιμή `new_val`. Αν η σύγκριση ήταν επιτυχής και η τιμή στη διεύθυνση μνήμης που δείχνει ο δείκτης `addr` έχει αλλάξει, επιστρέφεται μη μηδενική τιμή.

- `AO_TS_VAL_t test_and_set(volatile AO_TS_t * addr)`
  Ατομική ανάγνωση της δυαδικής τιμής που δείχνει ο δείκτης `addr` και ορισμός της.

- `int compare_double_and_swap_double(volatile AO_double_t * addr, AO_t old_val1, AO_t old_val2, AO_t new_val1, AO_t new_val2);`

- `int compare_and_swap_double(volatile AO_double_t * addr, AO_t old_val1, AO_t new_val1, AO_t new_val2);`

  Για συστήματα που δεν υποστηρίζουν την εντολή `compare_and_swap` για τιμές μεγαλύτερου έβρους (double), υπάρχουν αυτές οι δύο εντολές. Αυτές οι εντολές χωρίζουν την τιμή (double) σε ζευγάρια.

Συνοψίζοντας, οι έτοιμες βιβλιοθήκες παρέχουν μια πολύ καλή λύση για ατομικές εντολές. Κύριο μειονέκτημα τους όμως είναι η αμφίβολη μελλοντική αναβάθμιση με την εμφάνιση νέων επεξεργαστών. Κανείς δεν εγγυάται τον χρόνο ζωής μιας βιβλιοθήκης. Επιπλέον αν ενδιαφερόμαστε για την βέλτιστη απόδοση στην εφαρμογής, θα πρέπει να μας προβληματίσει ο κώδικας υλοποίησης των ατομικών εντολών. Στις έτοιμες βιβλιοθήκες δεν μπορούμε να γνωρίζουμε τους αλγόριθμους κρύβονται πίσω από κάθε ατομική εντολή.

3.4.3 GCC & Ατομικές Εντολές

Μια ενδιάμεση λύση από τις προαναφερόμενες είναι οι ατομικές εντολές που βρίσκονται ενσωματωμένες στον μεταγλωττιστή (compiler) GCC.

Ο GCC είναι ίσως ο δημοφιλέστερος μεταγλωττιστής ανοικτού κώδικα για UNIX & Windows λειτουργικά συστήματα και συντάσσεται από τα ακρώνυμα του GNU Compliler Collection. Από την έκδοση 4.2 και έπειτα ενσωματώθηκε στον κώδικά του ένα σύνολο από βασικές ατομικές εντολές σαν κλήσεις συναρτήσεων που μπορούν να κληθούν απευθείας από οποιαδήποτε εφαρμογή. Οι εντολές είναι οι ακόλουθες:

- `type __sync_fetch_and_add (type *ptr, type value, ...)`

- `type __sync_fetch_and_sub (type *ptr, type value, ...)`

- `type __sync_fetch_and_or (type *ptr, type value, ...)`

- `type __sync_fetch_and_and (type *ptr, type value, ...)`
- `type __sync_fetch_and_xor (type *ptr, type value, ...)`
- `type __sync_fetch_and_nand (type *ptr, type value, ...)`

Οι παραπάνω ατομικές εντολές παίρνουν την τιμή που βρίσκεται στη θέση μνήμης που δείχνει ο δείκτης `ptr` και ανάλογα με το όνομα της εντολής εκτελούν την κατάλληλη πράξη με την τιμή `value`. Το αποτέλεσμα το αποθηκεύουν πίσω στη θέση μνήμης και επιστρέφουν την τιμή πριν από την αλλαγή.

- `type __sync_add_and_fetch (type *ptr, type value, ...)`
- `type __sync_sub_and_fetch (type *ptr, type value, ...)`
- `type __sync_or_and_fetch (type *ptr, type value, ...)`
- `type __sync_and_and_fetch (type *ptr, type value, ...)`
- `type __sync_xor_and_fetch (type *ptr, type value, ...)`
- `type __sync_nand_and_fetch (type *ptr, type value, ...)`

Αυτή η ομάδα ατομικών εντολών είναι παρόμοια με την προηγούμενη με τη διαφορά ότι επιστρέφεται η τιμή στη θέση μνήμης μετά την αλλαγή.

- `bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval, ...)`
- `type __sync_val_compare_and_swap (type *ptr, type oldval, type newval, ...)`

Οι εντολές αυτές είναι παρόμοιες με της εντολές μηχανής CMPXCHG (Intel) και Compare and Swap (SUN). Παρέχουν ατομική σύγκριση και ανταλλαγή, δηλαδή αν η τιμή στη θέση μνήμης που δείχνει ο δείκτης `ptr` είναι ίση με τη `oldval`, τότε η `newval` εκχωρείται στην θέση μνήμης που δείχνει ο δείκτης. Η πρώτη εντολή (bool) επιστρέφει true αν η σύγκριση ήταν επιτυχής και η `newval` έχει εγγραφεί με επιτυχία στη μνήμη. Η δεύτερη εντολή (type) επιστρέφει την τιμή στη θέση μνήμης πριν από την αλλαγή.

- `__sync_synchronize (...)`

Αυτή η εντολή υλοποιεί έναν Memory Barrier.

- `__sync_lock_test_and_set (type *ptr, type value, ...)`

Η ατομική αυτή εντολή είναι παρόμοια με την XCHG (Intel) και SWAP (Sun). Ανταλλάσσει την τιμή στη θέση μνήμης που δείχνει ο δείκτης ptr με τη value και επιστρέφει την τιμή που βρισκόταν στη μνήμη προηγουμένως. Σε ορισμένα συστήματα αυτή η εντολή μπορεί να χρησιμοποιηθεί στην υλοποίηση κλειδαριών θέτοντας την τιμή `value = 1`.

- `void __sync_lock_release (type *ptr, ...)`

Αυτή η ατομική εντολή είναι η αντίστοιχη της `__sync_lock_test_and_set (type *ptr, type value, ...)` για απελευθέρωση της κλειδαριάς. Δηλαδή θέτει στη θέση μνήμη που δείχνει ο δείκτης `ptr` το 1.

Ο GCC απέδειξε ότι είναι ένα εργαλείο που θα παραμείνει για πολύ καιρό ακόμη στο χώρο των μεταγλωττιστών. Η συνεχής υποστήριξη, η κάλυψη σχεδόν όλων των υπάρχουσων αρχιτεκτονικών, η ευρεία χρήση και οι δυνατότητες που παρέχει τον καθιστούν πολύ δημοφιλή και ανταγωνιστικό σε σχέση με πολλούς εμπορικούς μεταγλωττιστές. Πέραν της μελλοντικής πορείας του, οι ατομικές εντολές που παρέχει αναγνωρίζονται αυτόματα και δεν χρειάζεται οποιαδήποτε προσθήκη βιβλιοθήκης κατά την μετάφραση. Επίσης ένα σοβαρό πλεονέκτημα είναι η συμβατότητά του με τον εξίσου δημοφιλή μεταγλωττιστή της Intel, ICC. Ενώ και άλλοι εμπορικοί μεταφραστές τείνουν να διατηρήσουν συμβατότητα με τον GCC. Αυτό έχει ως συνέπεια, ανεξάρτητα με τον πιο μεταγλωττιστή έχει ο χρήστης, ο κώδικας να είναι συμβατός προς μετάφραση.

Κεφάλαιο 4

OMPi Compiler & Ατομικές Εντολές

4.1 OMPi – Μια σύντομη περιγραφή

Ο OMPi είναι ένας «ελαφρύς», ανοικτού κώδικα μεταγλωττιστής για τη γλώσσα προγραμματισμού C, βασισμένος στο OpenMP 2.5. Είναι μεταφέρσιμος σε όλες τις πλατφόρμες που υποστηρίζουν POSIX threads αλλά μπορεί να υποστηρίξει οποιαδήποτε άλλη βιβλιοθήκη νημάτων.

Ο OMPi αποτελείται από δύο κύρια μέρη:

- Το μεταφραστή (compiler)
- τη βιβλιοθήκη υποστήριξης εκτέλεσης (runtime library)

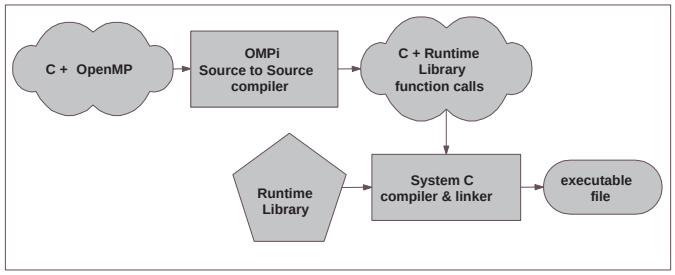
Ο μεταφραστής είναι ένας source–to–source compiler δηλαδή παίρνει κώδικα γραμμένο σε C με κλήσεις OpenMP (#pragma ...) και τον μετατρέπει σε κώδικα C με κλήσεις στη βιβλιοθήκη. Η βιβλιοθήκη αποτελείται και αυτή από δύο μέρη:

- Τον Ελεγκτή εκτέλεσης προγράμματος (Runtime Controler – ort.c)
- Την Βιβλιοθήκη νημάτων/διεργασιών (Thread/Process Library – othr.c)

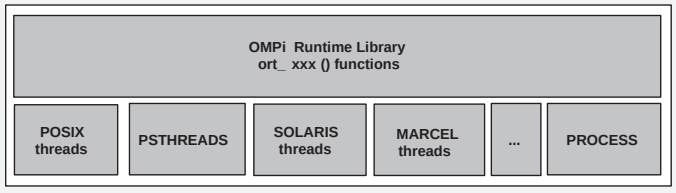
Κύρια εργασία του ελεγκτή είναι να παρέχει τις απαραίτητες πληροφορίες στα νήματα/διεργασίες για να μπορέσουν να εκτελεστούν. Για παράδειγμα, διαμοιράζει στα νήματα/διεργασίες το σύνολο των επαναλήψεων από ένα parallel for loop ή καθορίζει ποιος θα εκτελέσει το κάθε section.

Η βιβλιοθήκη νημάτων/διεργασιών δημιουργεί το σύνολο των νημάτων ή διεργασιών κατόπιν επιλογής του ελεγκτή και τα παραχωρεί σε αυτόν για διαχείριση. Επίσης δημιουργεί τις κλειδαριές και τον κοινό χώρο διευθύνσεων που θα προσπελάσουν οι διεργασίες.

Εδώ να διευκρινίσουμε ότι τα δυο μέρη της βιβλιοθήκης λειτουργούν ανεξάρτητα. Δηλαδή κάποιος μπορεί να δημιουργήσει τη δική του βιβλιοθήκη νημάτων και να την προσθέσει στον OMPi, εξού και το πλήθος των διαφορετικών βιβλιοθηκών νημάτων που υπάρχουν προεγκατεστημένα (pthreads, pthreads1, solthr, solthr1, psthreads, marcel).



Σχήμα 4.1: Η βασική δομή του OMPi.



Σχήμα 4.2: Δομή Βιβλιοθήκης.

4.2 Βελτιστοποιήσεις στη Βιβλιοθήκη

Η βιβλιοθήκη του OMPi, ως κύριος διαχειριστής των νημάτων/διεργασιών, πρέπει να υλοποιηθεί με τέτοιο τρόπο, ούτως ώστε να μην επιβαρύνει την ταχύτητα εκτέλεσης της εφαρμογής. Ο κώδικάς της θα πρέπει να είναι έξυπνος με όσο το δυνατό λιγότερες κλειδαριές (lock–free) και σημεία συγχρονισμού των νημάτων.

Σε μια προσπάθεια βελτίωσης του OMPi, εντάσσουμε σε ορισμένα σημεία του κώδικα ατομικές εντολές για να περιορίσουμε την χρήση κλειδαριών. Αργότερα ελέγχουμε αν όντως παρουσιάζεται κάποια βελτίωση στο χρόνο εκτέλεσης της εφαρμογής και καταλήγουμε σε κάποια βασικά συμπεράσματα. Κυρίως για λόγους απλότητας και μελλοντικής υποστήριξης, θα χρησιμοποιήσουμε την τρίτη λύση για τις ατομικές εντολέσ(παράγραφος 4.3), δηλαδή τις παρεχόμενες συναρτήσεις από τον μεταφραστή GCC.

Όπως προαναφέρθηκε, οι ατομικές εντολές παρουσιάζουν κάποιους περιορισμούς στη σωστή εφαρμογή τους. Μπορούν να τροποποιήσουν ατομικά και με ασφάλεια μονό μια θέση μνήμης. Με αυτόν τον περιορισμό πρέπει να αναζητήσουμε σημεία στον κώδικα που τροποποιούν μια κοινή μεταβλητή (μια θέση μνήμης) και το επιτυγχάνουν με αμοιβαίο αποκλεισμό υλοποιημένο με κλειδαριές.

Η βιβλιοθήκη Runtime του OMPi είναι ήδη βελτιστοποιημένη σε πολλά σημεία προκειμένου να επιτύχει υψηλές επιδόσεις, περιορίζοντας πολύ την χρήση κλειδαριών. Έτσι τα σημεία στα οποία μπορούμε να παρέμβουμε για να εισάγουμε ατομικές λειτουργίες είναι περιορισμένες. Παρόλα αυτά, τα σημεία αυτά αποδεικνύονται εξαιρετικά σημαντικά μιας και συνεισφέρουν στον συνολικό χρόνο εκτέλεσης μιας εφαρμογής όπως θα δούμε και στο επόμενο κεφάλαιο.

4.2.1 Αντιγραφη ιδιωτικων μεταβλητων μεταξυ νηματων

: Το πρώτο σημείο στον κώδικα που μπορεί να τροποποιηθεί είναι στη συνάρτηση `ort_copy_private()`. Η συνάρτηση αυτή αντιγράφει τις τιμές των ιδιωτικών μεταβλητών ενός νήματος, στα υπόλοιπα νήματα της ομάδας.

Καταρχήν η συνάρτηση αυτή προκύπτει στον παραγόμενο κώδικα από τη φράση `copyprivate(a, b, c)`, όπου `a, b, c` οι μεταβλητές οι οποίες η τιμή τους θα αντιγραφή στα υπόλοιπα νήματα.

Η φράση αυτή εμφανίζεται μόνο στην εντολή `#pragma omp single`.

Ένα παράδειγμα φαίνεται πιο κάτω:

```
#pragma omp single copyprivate (a, b, c)
  {
     a=1;
     b=2;
     c=3;
  }
```

Ο κώδικας που παράγεται από τον μεταφραστή έχει ως εξής:

```
/* #pragma omp single copyprivate(a, b, c) */
if (ort_mysingle(0))
  {
    {
       a=1;
       b=2;
       c=3;
    }
    ort_broadcast_private(3, &a, &b, &c);
  }
ort_leaving_single();
ort_barrier_me();
ort_copy_private(3, &a, sizeof(a), &b, sizeof(b), &c, sizeof(c));
ort_barrier_me();
```

Οι δείκτες των μεταβλητών που αντιγράφονται είναι αποθηκευμένοι σε ένα πίνακα δεικτών. Αυτός ο πίνακας δεσμεύεται με `malloc()` και ενημερώνεται στη συνάρτηση `ort_broadcast_private()` από το νήμα κάτοχο των μεταβλητών. Έπειτα μετά την έξοδο από την single περιοχή καλείται η `ort_copy_private()` από κάθε νήμα, και πραγματοποιείται η αντιγραφή.

Ο πίνακας που δεσμεύτηκε πρέπει με κάποιο τρόπο να αποδεσμευτεί. Για αυτό τον σκοπό υπάρχει μια κοινή μεταβλητή που μειώνεται κάθε φορά που ένα νήμα αντιγράφει τις μεταβλητές και εξέρχεται από τη συνάρτηση. Η μεταβλητή αυτή αρχικά έχει τιμή ίση με τον αριθμό νημάτων της ομάδας. Όταν γίνει 0 τότε συνεπάγεται ότι και το τελευταίο νήμα έχει αντιγράψει τα δεδομένα και ο πίνακας δεικτών μπορεί να αποδεσμευτεί. Με την έξοδο και του τελευταίου νήματος, ο πίνακας αποδεσμεύεται με `free()`. Η μεταβλητή στην αρχική έκδοση ενημερωνόταν ως εξής:

```
ee_set_lock(&cp->lock);
   i = --cp->copiers;     /* shared variable cp->copiers */
ee_unset_lock(&cp->lock);

if (i == 0) /* i local variable */
   free(cp->data);   /* Free allocated data */
```

Εύκολα μπορούμε να αντικαταστήσουμε αυτόν το κώδικα με μία ατομική εντολή, αποφεύγοντας τη χρήση κλειδαριών. Ο νέος κώδικας είναι:

```
1  i = __sync_fetch_and_add(&(cp->copiers), -1) - 1;
2
3  if (i == 0)      /* i local variable */
4     free(cp->data);  /* Free allocated data */
```

Παρατηρούμε ότι χρησιμοποιήθηκε η ατομική εντολή `fetch_and_add()`, ενώ μπορούσαμε να χρησιμοποιήσουμε την `add_and_fetch()` χωρίς να χρειαστεί να αφαιρέσουμε 1. Αυτό γίνεται για λόγους καλύτερης συμβατότητας με ατομικές εντολές επεξεργαστών μιας και οι περισσότεροι επεξεργαστές υποστηρίσουν εγγενώς την πρώτη μορφή και όχι την δεύτερη. Προσέξτε ότι η κοινή μεταβλητή `cp->data` φαίνεται ότι μειώνεται δυο φορές κατά 1. Αυτό γίνεται γιατί η συνάρτηση `__sync_fetch_and_add()` επιστέφει την τιμή πριν από την αλλαγή. Έτσι για να αποθηκευτεί η σωστή τιμή στην μεταβλητή `i` αφαιρείται η μονάδα.

4.2.2 Εισοδος σε περιοχη διαμοιρασμου εργασιας

Ένα δεύτερο σημείο στον κώδικα που μπορεί να τροποποιηθεί βρίσκεται στη συνάρτηση `leave_workshare_region()`. Στις περιοχές διαμοιρασμού εργασίας, τα νήματα, διαμοιράζονται την εργασία. Δηλαδή εκτελούν όλα ένα μέρος της εργασίας, μέχρι την πλήρη ολοκλήρωσή της. Οι περιοχές διαμοιρασμού εργασίας(`workshare_regions`) στο πρότυπο OpenMP είναι οι for, sections, και single.

Κάθε νήμα που εισέρχεται σε μια περιοχή `enter_workshare_region()` ελέγχει κατά πόσο μπορεί να την εκτελέσει η όχι. Αυτός ο έλεγχος γίνεται βάση μιας σταθεράς που δείχνει το μέγιστο επιτρεπτό όριο των ενεργοποιημένων περιοχών. Το πρώτο νήμα που θα εκτελέσει την περιοχή αρχικοποιεί ορισμένες μεταβλητές που αφορούν τον τύπο της περιοχής (for, sections, single) και τις κοινές μεταβλητές `empty`, `headregion` και `remain`. Η μεταβλητή `empty` γίνεται ίση με 0 όταν η περιοχή εκτελείται από κάποιο νήμα, και η `remain` τίθεται ίση με τον αριθμό των νημάτων που θα εκτελέσουν την περιοχή. Επίσης, η μεταβλητή `headregion` αυξάνεται κατά 1. Η `headregion` αυξάνεται όταν κάποια περιοχή ενεργοποιείται από κάποιο νήμα. Αντίστοιχη της `headregion` είναι η `tailregion` που με τη σειρά της αυξάνεται στο τέλος εκτέλεσης μιας περιοχής. Η διαφορά των δύο αυτών μεταβλητών μας δίνει τον αριθμό των περιοχών που είναι ενεργοποιημένοι. Βάση της διαφοράς αυτής, κάθε νήμα ελέγχει αν μπορεί να εκτελέσει μια περιοχή διαμοιρασμού εργασίας ή όχι. Κατά την έξοδο από την περιοχή κάθε νήμα μειώνει την `remain` κατά 1 και ελέγχει αν είναι το τελευταίο νήμα που εγκαταλείπει την περιοχή. Αν ναι, τότε θέτει το `empty` ίσο με 1 και αυξάνει το `tailregion`. Ο πιο κάτω κώδικας δείχνει ένα μέρος της διαδικασίας εξόδου (`leave_workshare_region()`) που μπορεί με ευκολία να τροποποιηθεί.

```
1  ee_set_lock(&r->reglock);
2     remain=--(r->notleft);   /*Increment the # threads that left*/
3     if (!remain)             /* If I am the last to leave */
4     {
5        r->empty = 1;         /* The region is now empty */
6        if (me->nowaitregion)
7          (ws->tailregion)++; /* Advance the tail */
8     }
9  ee_unset_lock(&r->reglock);
```

Ο κώδικας με ατομικές εντολές μετατρέπεται ως ακολούθως.

```
remain = __sync_fetch_and_add(&(r->notleft), -1) - 1;

if (!remain)              /* If I am the last to leave */
{
   r->empty = 1;          /* The region is now empty */
   if (me->nowaitregion)
      (ws->tailregion)++; /* Advance the tail */
}
```

Εδώ παρατηρούμε ότι η συνθήκη (if) δεν βρίσκεται εντός της κρίσιμης περιοχής. Το ίδιο θα μπορούσαμε να κάνουμε και στον κώδικα με τις κλειδαριές. Σε αυτή την περίπτωση όμως θα χρειαζόταν να βάλουμε στον κώδικα ένα Memory Barrier (FENCE) για να σιγουρευτούμε ότι σίγουρα έχουν γίνει οι αλλαγές στη μνήμη. Γενικά, ενημέρωση της μνήμης είναι μια πολύ χρονοβόρα διαδικασία που πρέπει να αποφεύγεται όπου είναι δυνατόν. Έτσι προτιμήθηκε να βρίσκεται όλος ο κώδικας στην κρίσιμη περιοχή για αποφυγή επιπλέον καθυστερήσεων.

4.2.3 Ανταγωνισμος για εκτελεση τμηματος κωδικα Section

Ένα τρίτο σημείο στον κώδικα που μπορεί να γίνει με ατομικές εντολές είναι εντός της συνάρτησης `ort_get_section()`. Σκοπός αυτής της συνάρτησης είναι να διαμοιράζει τις περιοχές Sections στα νήματα. Ο αρχικός κώδικας είναι ο εξής:

```
#pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        printf("Hello from thread %d\n", omp_get_thread_num());
        #pragma omp section
        printf("Hello from thread %d\n", omp_get_thread_num());
    }
```

Ο κώδικας που παράγεται από τον μεταφραστή έχει ως εξής:

```
/*(16)#pragma omp parallel num_threads(2) -- body moved below */
  {
    /* #pragma omp sections  */
    int caseid_ = -1, inpar_;

    if ((inpar_ = (omp_in_parallel() &&
                   omp_get_num_threads() > 1)) != 0)
      ort_entering_sections(1, 2);
    for (;;)
    {
      if (inpar_)
        {
          if ((caseid_ = ort_get_section()) < 0)
            break;
        }
      else
```

```
17          {
18            if ((++caseid_) >= 2)
19              break;
20          }
21        switch (caseid_)
22        {
23          case 0 :
24             printf("Hello from thread %d\n", omp_get_thread_num());
25             break;
26          case 1 :
27             printf("Hello from thread %d\n", omp_get_thread_num());
28             break;
29        }
30      }
31      if (inpar_)
32        ort_leaving_sections();
33    }
```

Όπως φαίνεται και στον πιο πάνω κώδικα η συνάρτηση ort_get_section() καλείται από κάθε νήμα και επιστρέφει τον αριθμό του Section που θα εκτελεστεί, στο νήμα που την κάλεσε. Αυτό γίνεται εύκολα με μια κοινή μεταβλητή που διατηρεί το συνολικό αριθμό των περιοχών Section και μειώνεται κατά ένα όταν δοθεί κάποια περιοχή προς εκτέλεση. Ο κώδικας εντός της βιβλιοθήκης Runtime είναι ο ακόλουθος.

```
1  ee_set_lock(&r->reglock);
2     s = --(r->sectionsleft);
3  ee_unset_lock(&r->reglock);
```

Οι τρεις γραμμές κώδικα μετατρέπονται εύκολα σε μια. Είναι ίδια με την περίπτωση στη συνάρτηση ort_copy_private().

```
1
2  s = __sync_fetch_and_add (&(r->sectionsleft), -1) - 1;
3
```

4.2.4 Σειριακη εκτελεση επαναληψεων μεσω Ordered

Η επόμενη συνάρτηση που μπορεί να τροποποιηθεί είναι η ort_ordered_end(). Η οδηγία ordered καθορίζει ότι οι επαναλήψεις του εσώκλειστου βρόγχου θα εκτελεστούν με τη σειρά όπως θα εκτελούνταν σε ένα σειριακό υπολογιστή. Το παράδειγμα που ακολουθεί παρουσιάζει τη χρήση μιας ordered οδηγίας.

```
1  #pragma omp parallel
2     {
3        #pragma omp for ordered
4        for (i = 0 ; i < 30 ; i++)
5           #pragma omp ordered
6           printf("%d   %d\n",i, omp_get_thread_num());
7     }
```

Ο κώδικας που παράγεται από τον μεταφραστή έχει ως εξής:

```
/* #pragma omp for ordered */
   int i;
   int from_ = 0, to_ = 0, step_;
   struct _ort_gdopt_ gdopt_;

   step_ = 1;
   ort_entering_for(1, 1, 0, step_, &gdopt_);
   if (ort_get_static_default_chunk(0, 30, step_, &from_, &to_))
     {
       ort_thischunk_lb(from_);
       for (i = from_; i < to_; i = i + 1)
         {
           /* #pragma omp ordered  */
           ort_ordered_begin();
           printf("%d   %d\n", i, omp_get_thread_num());
           ort_ordered_end();
         }
     }
   ort_leaving_for();
```

Όταν ένα νήμα το οποίο πρόκειται να εκτελέσει την πρώτη επανάληψη του βρόγχου, συναντήσει μια οδηγία ordered, τότε προχωράει στην εκτέλεση της ordered περιοχή χωρίς να περιμένει. Αντίθετα, όταν νήματα που εκτελούν επόμενες επαναλήψεις, συναντήσουν την ίδια ordered οδηγία, τότε περιμένουν στην αρχή της ordered περιοχής μέχρι να τελειώσουν από αυτή την περιοχή όλα τα νήματα που εκτελούν προηγούμενες επαναλήψεις (έτσι, μόνο ένα νήμα επιτρέπεται να είναι σε μια ordered περιοχή κάθε φορά). Αυτό επιτυγχάνεται με τη βοήθεια μιας κοινής μεταβλητής (o−>**next_iteration**). Το κάθε νήμα ελέγχει αν η επανάληψη που θα εκτελέσει είναι ίση με αυτή την μεταβλητή, και αν ναι, τότε προχωρά στην εκτέλεση. Διαφορετικά περιμένει μέχρι να έρθει η σειρά του.

Κάθε νήμα που εξέρχεται από μια ordered περιοχή αυξάνει την κοινή μεταβλητή κατά το βήμα επανάληψης για να μπορέσει το επόμενο προγραμματισμένο νήμα να την εκτελέσει. Η αύξηση της κοινής μεταβλητής φαίνεται πιο κάτω.

```
ee_set_lock(&o->lock);
   o->next_iteration += o->incr;
ee_unset_lock(&o->lock);
```

Η τροποποίηση με ατομικές εντολές είναι ίδια με τις προηγούμενες περιπτώσεις. Η διαφορά που παρατηρείται είναι η απουσία οποιασδήποτε προσθαφαίρεσης στην επιστροφή της συνάρτησης, αφού δεν χρησιμοποιείται η τιμή κάπου αλλού.

```

__sync_fetch_and_add(&(o->next_iteration), o->incr );

```

4.2.5 Δρομολογηση επαναληψεων με πολιτικη Dynamic

Η συνάρτηση ort_get_dynamic_chunk() είναι υπεύθυνη για τη διανομή του αριθμού επαναλήψεων (chunk = κομμάτι) που θα εκτελέσει κάθε νήμα σε ένα for loop. Η διανομή αυτή γίνεται δυναμικά, όταν ένα νήμα τελειώσει θα αναλάβει δυναμικά κάποιο άλλο κομμάτι. Το παράδειγμα που ακολουθεί παρουσιάζει ένα βρόγχο με πολιτική Dynamic.

```
1  #pragma omp parallel
2     {
3        #pragma omp for schedule(dynamic)
4        for (i = 0 ; i < 30 ; i++)
5           printf("%d   %d\n",i, omp_get_thread_num());
6     }
```

Ο κώδικας που παράγεται από τον μεταφραστή έχει ως εξής:

```
1  /* #pragma omp for schedule(dynamic ) */
2       int i;
3       int from_ = 0, to_ = 0, step_;
4       struct _ort_gdopt_ gdopt_;
5
6       step_ = 1;
7       ort_entering_for(1, 0, 0, step_, &gdopt_);
8       while (ort_get_dynamic_chunk
9             (0, 30, step_, 1, &from_, &to_, (int *) 0, &gdopt_))
10       {
11          for (i = from_; i < to_; i = i + 1)
12             printf("%d   %d\n", i, omp_get_thread_num());
13       }
14      ort_leaving_for();
```

Για να μάθει κάποιο νήμα το τμήμα επαναλήψεων που θα εκτελέσει θα πρέπει να γνωρίζει τον αριθμό των επαναλήψεων που έχουν ήδη εκτελεστεί. Ο αριθμός των επαναλήψεων αποθηκεύεται σε μια κοινή μεταβλητή εντός της συνάρτησης ort_get_dynamic_chunk(). Αυτή η κοινή μεταβλητή (t−>data) ενημερώνεται όταν ανατεθεί σε κάποιο νήμα ένας αριθμός επαναλήψεων, με την πρόσθεση αυτού του αριθμού επαναλήψεων (step*chunksize) στη μεταβλητή. Έτσι μπορούμε αυτή την αύξηση να την υλοποιήσουμε με ατομικές εντολές αντί με τη χρήση κλειδαριών. Ο κώδικας που εξηγήθηκε είναι ο ακόλουθος.

```
1  ee_set_lock((ee_lock_t *) t->lock);
2     newlb = *(t->data);
3     (*(t->data)) += step*chunksize;
4  ee_unset_lock((ee_lock_t *) t->lock);
```

Η μετατροπή του κώδικα είναι

```
1
2  newlb = __sync_fetch_and_add(t->data , step*chunksize );
3
```

4.2.6 Δρομολογηση επαναληψεων με πολιτικη GUIDED

Η τελευταία βελτίωση που έγινε, παρουσιάζει ένα βαθμό δυσκολίας μεγαλύτερο από τις υπόλοιπες και βρίσκεται στη συνάρτηση `ort_ get_guided_chunk()`. Η συνάρτηση αυτή διανέμει στα νήματα τον αριθμό των επαναλήψεων που θα εκτελέσουν. Η διαφορά σε σχέση με τη συνάρτηση `ort_get_dynamic_chunk()` είναι το μέγεθος του κομματιού. Αντίθετα με την `ort_get_dynamic_chunk()` που το κομμάτι επανάληψης είναι σταθερό, της `ort_ get_guided_chunk()` μειώνεται δυναμικά. Το σκεπτικό, πίσω από αυτό το μηχανισμό είναι, τα "γρήγορα" νήματα να μπορέσουν να εκτελέσουν τις περισσότερες επαναλήψεις, ενώ τα πιο "αργά" να εκτελέσουν λιγότερες για να επιτευχθεί ισοζυγία φόρτου εκτέλεσης Load–Balancing. Αυτός ο τρόπος σε ορισμένες περιπτώσεις μπορεί να βελτιώσει πολύ το χρόνο εκτέλεσης. Ο κώδικας της εφαρμογής που τίθεται ως παράδειγμα είναι ο ίδιος όπως και πριν αλλά με πολιτική Guided αντί Dynamic. Έτσι ο κώδικας που παράγει ο μεταγλωττιστής είναι ο ακόλουθος.

```
1  /* #pragma omp for schedule(guided ) */
2      int i;
3      int from_ = 0, to_ = 0, step_;
4      struct _ort_gdopt_ gdopt_;
5
6      step_ = 1;
7      ort_entering_for(1, 0, 0, step_, &gdopt_);
8      while (ort_get_guided_chunk
9            (0, 30, step_, 1, &from_, &to_, (int *) 0, &gdopt_))
10      {
11         for (i = from_; i < to_; i = i + 1)
12            printf("%d   %d\n", i, omp_get_thread_num());
13      }
14     ort_leaving_for();
```

Όπως φαίνεται στον πιο πάνω κώδικα η συνάρτηση `ort_get_dynamic_chunk()` ενημερώνει τις τιμές των μεταβλητών `from`, `to` που καθορίζουν τα όρια που θα εκτελεστεί ο βρόγχος από το νήμα που την κάλεσε. Εντός αυτής της συνάρτησης ενημερώνεται και η κοινή μεταβλητή `*(t->data)` που με την βοήθειά της θα οριστούν οι τιμές `from` και `to`. Ένα μέρος του κώδικας της `ort_get_dynamic_chunk()` πριν από την τροποποίηση είναι ο ακόλουθος:

```
1  ee_set_lock((ee_lock_t *) t->lock);
2     newlb = *t->data;
3     d = div(ub - newlb, step * t->nth);
4     /* Remaining #iterations/#threads */
5     iters = d.quot;
6     if (d.rem != 0)
7        iters++;
8     if (iters > chunksize)
9        /* Give no less than chunksize iters */
10       chunksize = iters;
11    (*t->data) += step * chunksize;
12 ee_unset_lock((ee_lock_t *) t->lock);
```

Ο κώδικας αυτός υπολογίζει μέσα σε μια κρίσιμη περιοχή το σύνολο των επαναλήψεων (step * chunksize) που έχουν μέχρι τώρα εκτελεστεί και τα προσθέτει στην κοινή μεταβλητή `*(t->data)`.

Οι υπολογισμοί αυτοί πρέπει να εκτελεστούν ατομικά από κάθε νήμα, αλλά λόγω της πολυπλοκότητας τους δεν μπορούν να εκτελεστούν με μία ατομική εντολή αφού δεν είναι ένα απλό καθολικό άθροισμα. Βάση της τιμής της κοινής μεταβλητής υπολογίζεται το μέγεθος του επόμενου κομματιού. Βασικά αυτό που πρέπει να ελέγχουμε είναι ότι κανένα άλλο νήμα δεν έχει τροποποιήσει την κοινή μεταβλητή κατά την διάρκεια των ενδιάμεσων υπολογισμών. Αυτό μπορεί να γίνει με ένα do-while loop και τη βοήθεια της compare and swap όπως φαίνεται πιο κάτω.

```
1  do{
2     newlb = *t->data;
3     d = div(ub - newlb, step * t->nth);
4     iters = d.quot;
5     if (d.rem != 0)
6     iters++;
7     if (iters > chunksize)
8        chunksize = iters;
9    } while( !__sync_bool_compare_and_swap
10                    (t->data, newlb, newlb + step * chunksize) );
```

Ο πιο πάνω κώδικας φαίνεται ότι παρέχει αμοιβαίο αποκλεισμό με ένα κάπως ανορθόδοξο τρόπο σε σχέση με τις προηγούμενες περιπτώσεις. Για να δούμε πως ακριβώς λειτουργεί.

Έστω ένα νήμα εκτελεί τον πιο πάνω κώδικα. Η πρώτη εντολή που εκτελεί αφού μπει στο do – while loop είναι να αποθηκεύσει την κοινή μεταβλητή σε μια τοπική μεταβλητή, την `newlb`. Έπειτα εκτελεί κάποιους υπολογισμούς με τοπικές μεταβλητές και ακολούθως φτάνει στο while() με συνθήκη την τιμή επιστροφή της συνάρτησης `!__sync_bool_compare_and_swap(t->data, newlb, newlb + step * chunksize)`. Η συνάρτηση compare and swap πραγματοποιεί ατομική σύγκριση μεταξύ της τιμής `t->data` και `newlb` και αν είναι ίσες αντικαθιστά την τιμή που δείχνει ο δείκτης `t->data` με `newlb + step * chunksize`. Επίσης επιστέφει 1 αν η σύγκριση ήταν επιτυχής και η τιμή έχει αντιγραφεί κανονικά. Αυτό εγγυάται αμοιβαίο αποκλεισμό γιατί αν κάποιο νήμα είναι πιο γρήγορο από κάποιο άλλο και προλάβει να αλλάξει τη κοινή μεταβλητή `t->data` κατά την διάρκεια που το αργό νήμα είναι εντός του βρόγχου, τότε η συνάρτηση compare and swap για το αργό νήμα θα επιστέψει 0 και το do – while loop θα επαναληφθεί μέχρι να η τιμή `t->data` οριστεί σωστά.

Αυτός ο μηχανισμός φαινομενικά εμπεριέχει τον κίνδυνο καθυστέρησης. Ένα νήμα μπορεί να χρειαστεί να εκτελέσει περισσότερες από μία φορές το βρόγχο για να μπορέσει να τροποποιήσει σωστά την κοινή μεταβλητή. Παρόλα αυτά αυτή η λύση αποδεικνύεται ότι επιφέρει βελτίωση σε σχέση με τις κλειδαριές. Ο λόγος είναι ότι εφόσον δεν υπάρχει συναγωνισμός, μόνο μια φορά θα χρειαστεί να εκτελεστεί ο βρόγχος, κερδίζοντας χρόνο σε σχέση με τις κλειδαριές. Αν πάλι υπάρχει αρκετός ενεργός ανταγωνισμός δεν αναμένεται χειρότερη επίδοση από τις κλειδαριές διότι και στις κλειδαριές υπάρχει επαναληπτική προσπάθεια μέχρι το επιτυχημένο κλείδωμα.

Εδώ πρέπει να αναφερθεί ότι οι αλλαγές που έγιναν στον κώδικα δεν μπορούν να αντικαταστήσουν πλήρως τις κλειδαριές, και επομένως οι δύο λύσεις πρέπει να συνυπάρχουν. Αυτό χρειάζεται στην περίπτωση που στον υπολογιστή που θα εγκατασταθεί ο OMPi, ή η οποιαδήποτε εφαρμογή, δεν υποστηρίζονται ατομικές εντολές. Για παράδειγμα μπορεί να έχει έκδοση παλαιότερη από την 4.2 του GCC.

Στο επόμενο κεφάλαιο γίνεται αποτίμηση της βελτίωσης που προκύπτει από τις τροποποιήσεις μας στη βιβλιοθήκη υποστήριξης εκτέλεσης.

Κεφάλαιο 5

Αποτίμηση Επιδόσεων

Για την εκτίμηση της απόδοσης του OMPi έχουμε χρησιμοποιήσει δυο μετροπρογράμματα. Τα EPCC OpenMP Micro Benchmarks[3] [4] και τα NAS Parallel Benchmarks[6]. Τα δεύτερα αποτελούν ολοκληρωμένες εφαρμογές.

5.1 EPCC OpenMP Micro Benchmarks

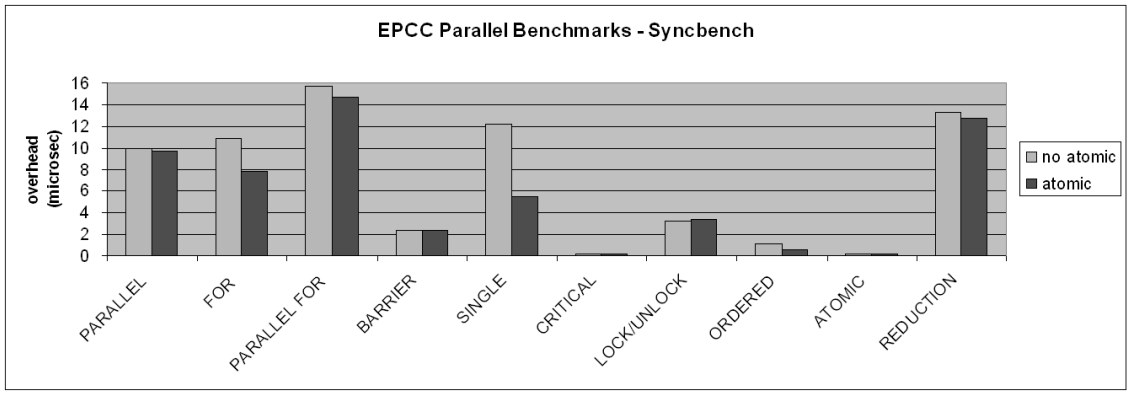
Τα EPCC OpenMP Micro Benchmarks είναι συνθετικές ρουτίνες για τη χρονομέτρηση παράλληλων περιοχών, περιοχών διαμοιρασμού εργασίας και των πλείστων εντολών του OpenMP για τη Runtime βιβλιοθήκη του compiler. Κάθε ρουτίνα βρίσκει τον μέσο χρόνο εκτέλεσης μιας περιοχής ή εντολής και το αντίστοιχο overhead. Επίσης μαζί με το αποτέλεσμα εμφανίζεται και το πιθανό σφάλμα. Χρησιμοποιούνται για την αποτίμηση των καθυστερήσεων που εισάγονται σε κάθε οδηγία OpenMP (και όχι για χρονομέτρηση εφαρμογών).

Η εφαρμογή εκτελεί τις μετρήσεις με δυο προγράμματα, το Syncbench και το Schedbench. Το Syncbench εκτελεί μετρήσεις στα PARALLEL, FOR, PARALLEL FOR, BARRIER, SINGLE, CRITICAL, LOCK/UNLOCK, ORDERED, ATOMIC, REDUCTION ενώ το Schedbench στον τρόπο εκτέλεσης του PARALLEL FOR με STATIC, DYNAMIC, και GUIDED με διαφορετικό κομμάτι επανάληψης (chunk size).

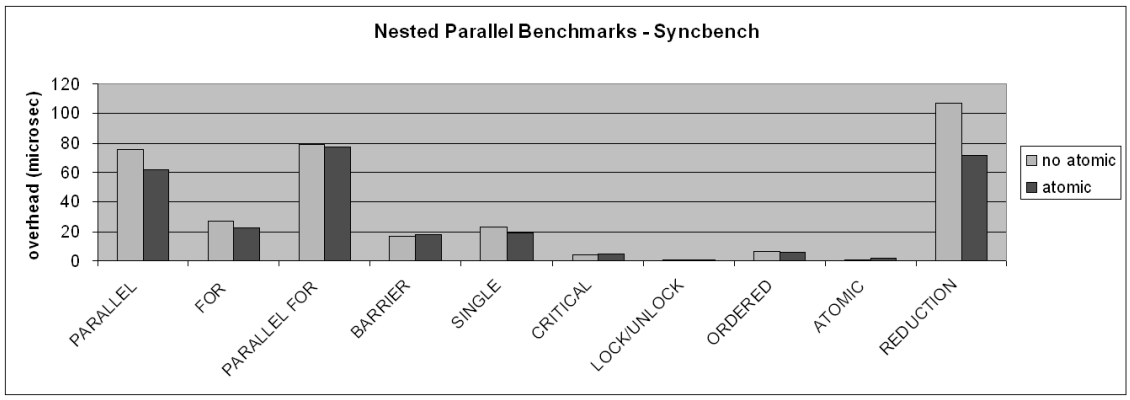
Εκτός από τα απλά EPCC OpenMP Micro Benchmarks υπάρχουν και τα Nested EPCC OpenMP Micro Benchmarks. Τα Nested EPCC OpenMP Micro Benchmarks [8] είναι μια τροποποίηση των απλών Benchmarks και μετρούν το Overhead σε εφαρμογές με εμφωλευμένα επίπεδα παραλληλίας.

Όλες οι μετρήσεις έγιναν σε ένα υπολογιστή SMP με 4 επεξεργαστές Intel Pentium III των 700MHz με συνολική μνήμη 1500Mb. Το λειτουργικό σύστημα του υπολογιστή είναι Debian GNU/Linux 4.0 με τον πυρήνα 2.6. Οι μετρήσεις έγιναν με 4 νήματα (όσοι και οι επεξεργαστές) σε περίοδο που η μηχανή είχε μηδενικό φόρτο εργασίας. Τα Nested Parallel Benchmarks εκτελέστηκαν με 16 νήματα (4x4). Η έκδοση του OMPi με την οποία πραγματοποιήθηκαν οι μετρήσεις είναι η 1.0.0 με βιβλιοθήκη νημάτων την pthreads.

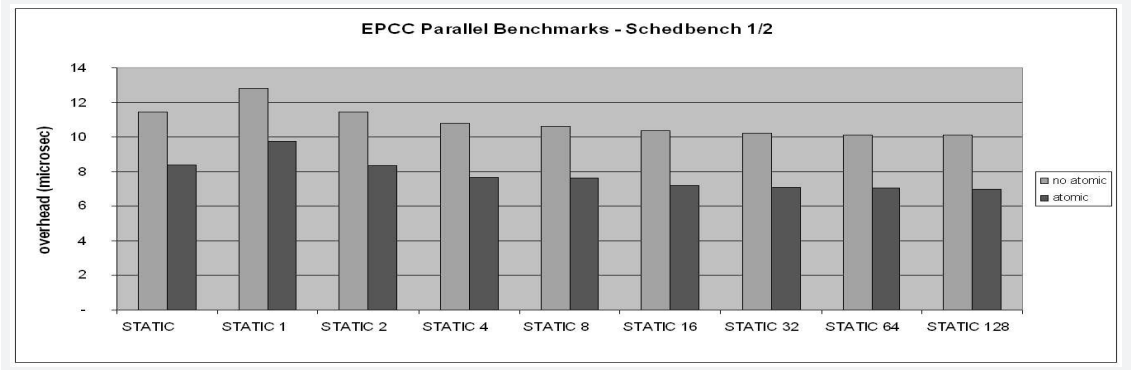
Οι γραφικές παραστάσεις από τις μετρήσεις των EPCC OpenMP Micro Benchmarks και Nested EPCC OpenMP Micro Benchmarks δίνονται στα σχήματα 5.1 και 5.2 αντίστοιχα.



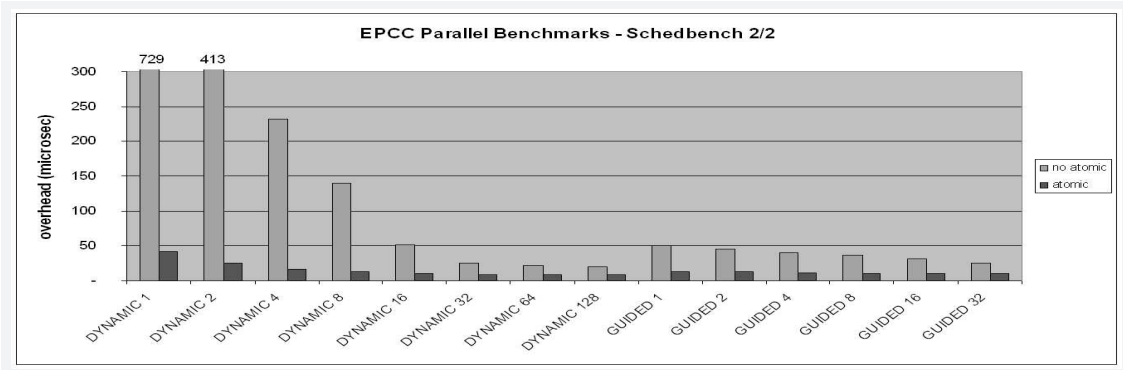
Σχήμα 5.1: Αποτελέσματα – Syncbench



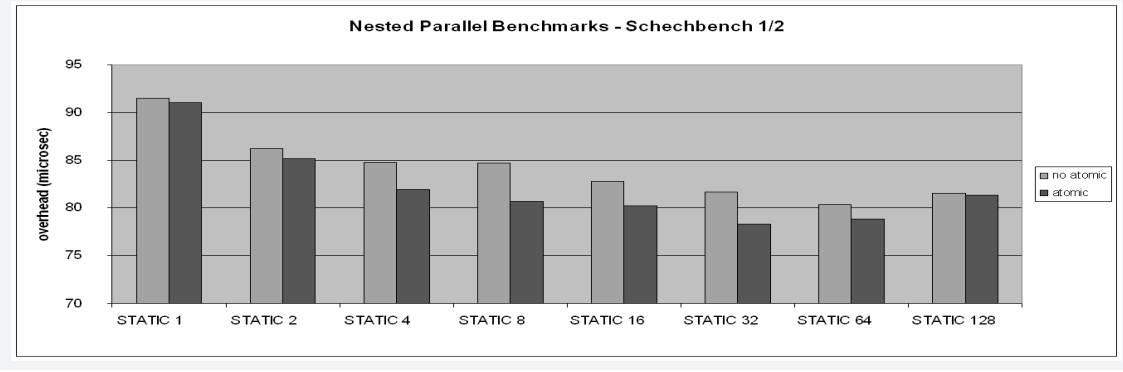
Σχήμα 5.2: Αποτελέσματα – Syncbench Nested



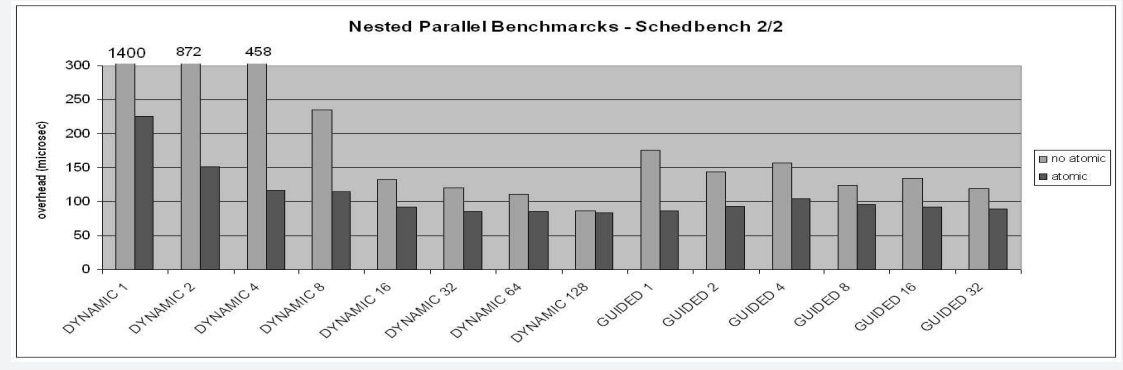
Σχήμα 5.3: Αποτελέσματα – Schedbench 1/2



Σχήμα 5.4: Αποτελέσματα – Schedbench 2/2



Σχήμα 5.5: Αποτελέσματα – Schedbench Nested 1/2



Σχήμα 5.6: Αποτελέσματα – Schedbench Nested 2/2

Όπως αναφέρθηκε και προηγουμένως για την δρομολόγηση στο Guided ο κώδικάς μας έκανε χρήση της CAS επαναληπτικά ώστε φαινομενικά να υπάρχει λόγος αμφιβολίας για το κέρδος σε απόδοση τις συνάρτησης ort_get_guided_chunk() με ατομικές εντολές. Οι πιο κάτω πίνακες παρουσιάζουν τα αποτελέσματα από το Schedbench και Schedbench Nested με και χωρίς τη χρήση της ατομικής εντολής (CAS – Compare and swap) στη συνάρτηση. Η πρώτη στήλη των πινάκων παρουσιάζει τους χρόνους εκτέλεσης χωρίς ατομικές εντολές. Η δεύτερη παρουσιάζει τους χρόνους με όλες τις ατομικές εντολές, ενώ η τρίτη χωρίς την ατομική εντολή CAS – Compare and swap εντός της συνάρτησης ort_get_guided_chunk(). Αυτές οι μετρήσεις έγιναν για να συγκριθούν οι χρόνοι ανάλογα με την περίπτωση και να δούμε αν υπάρχει όντος βελτίωση.

Schedbench	**No Atomic**	**Atomic & CAS**	**Atomic & No CAS**
GUIDED 1	49,960	12,457	43,063
GUIDED 2	44,949	12,278	38,538
GUIDED 4	40,686	11,595	33,923
GUIDED 8	36,875	10,247	29,385
GUIDED 16	31,773	9,557	24,418
GUIDED 32	25,701	9,792	19,102

Πίνακας 5.1: Αποτελέσματα Schedbench – ort_get_guided_chunk()

Schedbench Nested	**No Atomic**	**Atomic & CAS**	**Atomic & No CAS**
GUIDED 1	175,192	86,255	164,182
GUIDED 2	142,997	93,057	136,502
GUIDED 4	156,552	104,035	144,734
GUIDED 8	124,863	95,666	140,723
GUIDED 16	133,977	92,117	108,942
GUIDED 32	118,285	88,968	124,101

Πίνακας 5.2: Αποτελέσματα Schedbench Nested – ort_get_guided_chunk()

Δυστυχώς τα EPCC OpenMP Micro Benchmarks δεν παρέχουν ρουτίνα για τη μέτρηση του overhead στα sections. Έτσι για να μετρηθούν οι χρόνοι στα sections αναπτύξαμε ένα μικρό πρόγραμμα για αυτό τον σκοπό. Ο κώδικας αποτελείται από ένα βρόγχο που εκτελείται 10000000 φορές και μέσα έχει ένα #pragma parallel section με 50 sections που το καθένα αυξάνει ατομικά μια μεταβλητή.

Ο ακόλουθος κώδικας παρουσιάζει τον τρόπο μέτρησης πολλαπλών Sections. Θα εκτελεστεί δύο φορές. Με ατομικές εντολές και χωρίς, για να δούμε τι χρόνους παίρνουμε από την τροποποίηση της ort_get_section(). Τα αποτελέσματα από τη χρονομέτρηση του βρόγχου παρουσιάζονται στον Πίνακα 5.3.

```
1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <unistd.h>
4  #include <omp.h>
5
6  main()
7  {
8    struct timeval start, finish;
9    int msec,i=0,j=0;
10
11   gettimeofday (&start, NULL);          /* time before for loop */
12
13   for(j=0;j<1000;j++)
14   {
15     #pragma omp parallel shared(i)
16     {
17       #pragma omp sections
18       {
19        #pragma omp section               /* number 1 section */
20           {__sync_fetch_and_add(&i,1);}

                 ⋮

97        #pragma omp section               /* number 50 section */
98           {__sync_fetch_and_add(&i,1);}
99       }
100    }
101  }
102
103  gettimeofday (&finish, NULL);          /* time after for loop */
104
105  msec = finish.tv_sec*1000 + finish.tv_usec/1000;
106  msec -= start.tv_sec*1000 + start.tv_usec/1000;
107
108  printf("i = %d, Time: %d milliseconds\n",i, msec);  /* resault */
109 }
```

Αριθμός Επαναλήψεων	**Atomic**	**No Atomic**
1000000	29.928 sec	114.644 sec
100000	3.069 sec	12.014 sec
10000	0.297 sec	1.115 sec
100	0.025 sec	0.116 sec

Πίνακας 5.3: Αποτελέσματα ort_get_section()

5.2 Ανάλυση Αποτελεσμάτων & Συμπεράσματα

Με μια πρώτη ματιά στις γραφικές παραστάσεις παρατηρούμε σημαντικές βελτιώσεις. Οι χρόνοι σχεδόν σε κάθε έλεγχο των EPCC OpenMP Micro Benchmarks έχουν μειωθεί με τη χρήση των ατομικών εντολών. Ας δούμε τώρα πιο προσεκτικά που μπορεί να οφείλονται αυτές οι βελτιώσεις.

Σε γενικές γραμμές η αποφυγή μια κλειδαριάς επιφέρει πάντοτε σημαντικές βελτιώσεις στην εκτέλεση μιας παράλληλης εφαρμογής, ιδιαίτερα στην περίπτωση που τα νήματα/διεργασίες φτάνουν συγχρονισμένα και προσπαθούν να διαβάσουν και να τροποποιήσουν μια κλειδαριά. Τότε δημιουργείται μεγάλη καθυστέρηση όπως και απώλεια επεξεργαστικής ικανότητας του υπολογιστή λόγο της ενεργού αναμονής (busy–waiting). Άρα η απόδοση της εφαρμογής θα αυξηθεί ανάλογα με την χρήση των ατομικών εντολών. Όσο πιο πολύ χρησιμοποιούνται τα σημεία του κώδικα με ατομικές εντολές τόσο θα μειώνεται και ο χρόνος εκτέλεσης.

Τροποποιηση 1^{η}: Αυτή η αλλαγή στην ort_copy_private() δυστυχώς δεν μπορεί να μετρηθεί από τα EPCC OpenMP Micro Benchmarks. Δύσκολα μπορούμε να την μετρήσουμε, αλλά σίγουρα κάποιο ελάχιστο ποσοστό βελτίωσης μπορεί να παρατηρηθεί. Για τις υπόλοιπες αλλαγές έχουμε πιο άμεσες και σίγουρες αποδείξεις για την απόδοσή τους.

Τροποποιηση 2^{η}: Όπως αναφέρθηκε το OpenMP έχει τρεις διαφορετικές περιοχές διαμοιρασμού εργασίας τις for, section και single. Ο OMPi διαχειρίζεται τις περιοχές αυτές με τη βοήθεια των συναρτήσεων enter_workshare_region() και leave_workshare_region() όπου η δεύτερη έχει τροποποιηθεί με ατομικές εντολές. Βάσει αυτής της αλλαγής αναμένουμε να δούμε διαφορά στους χρόνους των for, section και single. Όπως φαίνεται και από τις γραφικές παραστάσεις η μείωση αν και μικρή, υπάρχει. Κατά την έξοδο από την περιοχή διαμοιρασμού εργασίας τα νήματα κερδίζουν χρόνο κατά την τροποποίηση της κοινής μεταβλητής με την ατομική εντολή fetch and add.

Τροποποιηση 3^{η}: Όπως προανέφερα τα EPCC OpenMP Micro Benchmarks δυστυχώς δεν παρέχουν μετροπρογράμματα για την μέτρηση του Overhead για περιοχές Section. Βάση όμως των μετρήσεων στον Πίνακα 5.3 ο χρόνος έχει μειωθεί περίπου στο 26% του αρχικού. Άρα πράγματι υπάρχει μια πολύ μεγάλη διαφορά στο χρόνο εκτέλεσης με ατομικές εντολές.

Τροποποιηση 4^{η}: Η επόμενη αλλαγή πραγματοποιήθηκε στη συνάρτηση ort_ ortered_end() και άρα αναμένουμε βελτίωση στο χρόνο εκτέλεσης του Ordered. Οι περιοχές Ordered εκτελούνται σειριακά σαν να εκτελούνταν σε ένα σειριακό υπολογιστή. Παρατηρείται μια σημαντική βελτίωση του 50% με τη χρήση ατομικών εντολών σε σχέση με τις κλειδαριές. Επίσης παρουσιάζεται και κάποια μικρότερη βελτίωση στα Nested EPCC OpenMP Micro Benchmarks.

Τροποποιηση 5^{η}: Η 5η και πιο κερδοφόρα αλλαγή, απ 'οτι παρουσιάζεται στις γραφικές παραστάσεις, έγινε στην ort_get_dynamic_chunck(). Λόγω του μεγάλου αριθμού κλήσεων της συνάρτησης από τα νήματα για να πάρουν το κομμάτι επανάληψης τους (DYNAMIC 1) η κλειδαριά καθυστερούσε την τροποποίηση της κοινής μεταβλητής. Βάση των αποτελεσμάτων, τα νήματα πρέπει να έφταναν συγχρονισμένα στην κλειδαριά, και ως αποτέλεσμα παρουσιαζόταν μεγάλη καθυστέρηση. Τώρα με τις ατομικές εντολές, ως lock–free και wait–free συνάρτηση, τα νήματα δεν περιμένουν και κινούνται ελεύθερα. Το κέρδος που παρουσιάζεται για DYNAMIC 1 (μέγεθος κομματιού 1) φτάνει το εντυπωσιακό ποσοστό 1700% (730 πριν, και 42 μετά) και μειώνεται στο 250% (20 πριν,και 8 μετά) μέχρι το DYNAMIC 128. Επίσης και στα Nested Benchmarks παρουσιάζεται δραματική βελτίωση. Αυτό είναι ένα ουσιαστικό παράδειγμα για το κέρδος που μπορούμε να επιτύχουμε με τις ατομικές εντολές.

Τροποποιηση 6^{η}: Η 6η και τελευταία αλλαγή παρόλο που έγινε σε συνάρτηση παρόμοια με την ort_get_dynamic_chunck(), δηλαδή μια συνάρτηση που διανέμει στα νήματα το κομμάτι επανάληψης, δεν παρουσιάζει την ίδια βελτίωση. Αυτό οφείλετε στην υλοποίηση της ίδιας της συνάρτησης. Η

συνάρτηση ort_get_guided_chunck() μειώνει δυναμικά το κομμάτι επανάληψης. Αυτό συνεπάγεται ότι ο κόκκος της παραλληλία δεν είναι ομοιόμορφος με αποτέλεσμα τα νήματα να εκτελέσουν λιγότερες φορές τη συνάρτηση ort_get_guided_chunck() σε σχέση με τη ort_get_dynamic_chunck(). Αυτό δικαιολογεί τον μειωμένο χρόνο εκτέλεση της. Παρόλα αυτά έχουμε ένα σεβαστό ποσοστό μείωσης, με τη χρήση ατομικών εντολών που φτάνει το 50% σε καθε εκτέλεση του GUIDED 1-32, όπου 1-32 ο ελάχιστος κόκκος παραλληλίας. Το ίδιο ποσοστό περίπου παίρνουμε και για τα Nested Benchmarks.

Η βελτίωση όμως αυτή θα μπορούσε να προέρχεται μόνο από τις ατομικές εντολές εντός της leave_workshare_region() από την 2η τροποποίηση. Για να το ερευνήσουμε αυτό, πήραμε μετρήσεις με και χωρίς την ατομική εντολή στην ort_get_guided_chunck(), όπως επίσης και χωρίς καθόλου ατομικές εντολές. Τα αποτελέσματα φαίνονται στους πίνακες 5.1 & 5.2. Όπως παρατηρούμε μία μικρή μόνο βελτίωση οφείλεται από τις ατομικές εντολές στην συνάρτηση leave_workshare_region(). Το μεγαλύτερο ποσοστό οφείλεται στην compare and swap εντός της ort_get_guided_chunck() και σίγουρα επιφέρει ουσιαστική αλλαγή.

5.3 NAS Parallel Benchmarks

Τα NAS Parallel Benchmarks είναι ένα σύνολο από ολοκληρωμένες παράλληλες εφαρμογές με πολύπλοκους επιστημονικούς υπολογισμούς υλοποιημένες από τη NASA. Πιο γνωστές από τις εφαρμογές είναι ο τρισδιάστατος μετασχηματισμός Fourier και η λύση γραμμικού συστήματος με την μέθοδο LU. Το σύνολο των εφαρμογών παρουσιάζεται πιο κάτω.

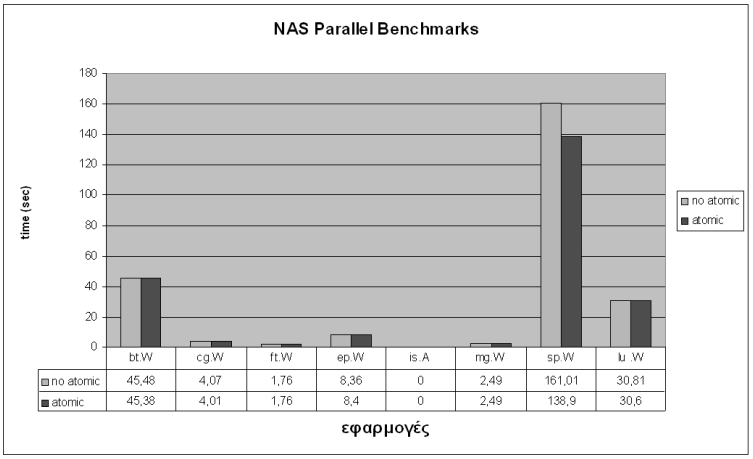
- Embarrassingly parallel (EP)
- Multigrid (MG)
- Conjugate gradient (CG)
- 3–D FFT PDE (FT)
- Integer sort (IS)
- LU solver (LU)
- Pentadiagonal solver (SP)
- Block tridiagonal solver (BT)

Αυτές οι παράλληλες εφαρμογές υποστηρίζουν αύξηση της πολυπλοκότητας τους. Δηλαδή μπορούν να τροποποιηθούν ανάλογα με το μηχάνημα που θα πραγματοποιηθεί η μέτρηση. Στην περίπτωση μας η κλάση πολυπλοκότητας που επιλέχθηκε είναι η W τύπου σταθμού εργασίας (Workstation). Η εφαρμογή IS λόγω του "μηδενικού" χρόνου εκτέλεσης της με κλάση W εκτελέστηκε με κλάση A που έχει τι μεγαλύτερη πολυπλοκότητα. Το αποτέλεσμα όμως παρέμεινε το ίδιο.

Με την εκτέλεση αυτών των εφαρμογών με και χωρίς ατομικές εντολές προέκυψαν κάποια πολύ ενδιαφέροντα αποτελέσματα. Η γραφική παράσταση των αποτελεσμάτων είναι η ακόλουθη.

Τα NAS Parallel Benchmarks ως πολύπλοκες υπολογιστικές εφαρμογές χρειάζονται πολύ χρόνο προτού ολοκληρώσουν την εκτέλεση τους. Για να υπάρξει κάποια ουσιαστική αλλαγή στο χρόνο εκτέλεσης από τις ατομικές εντολές θα πρέπει η εφαρμογή να περιέχει κώδικα ο οποίος οδηγεί έμμεσα σε κλήσεις της βιβλιοθήκης Runtime όπου έχουμε εισάγει ατομικές εντολές και κυρίως βρόγχους for με μη στατικά schedules.

Με βάση τις μετρήσεις που έχουμε, δεν βλέπουμε σε καμιά δραματική αλλαγή όσο αφορά το χρόνο, εκτός από μια εφαρμογή και λιγότερο σε άλλες τρις. Η εφαρμογή που παρουσιάζει την μεγαλύτερη βελτίωση είναι η SP, μείωση στο 86.26% του αρχικού χρόνου. Οι εφαρμογές BT, CG και LU



Σχήμα 5.7: Αποτελέσματα – NAS Parallel Benchmarks

παρουσιάζουν βελτίωση του $\approx 1\%$. Οι υπόλοιπες εφαρμογές αν παρουσιαζουν κάποια βελτίωση, είναι ελάχιστη και δεν μπορεί να μετρηθεί.

Γιατί όμως μερικές εφαρμογές να παρουσιάζουν βελτίωση και άλλες όχι. Αυτό εξηγείται από τον τον πιο κάτω πίνακα.

NAS Benchmarks	**# for**	**# for nowait**
BT	38	16
CG	17	4
FT	5	1
EP	1	0
IS	0	1
MG	11	0
SP	62	8
LU	22	7

Πίνακας 5.4: NAS – Αριθμός #pragma for και #pragma for nowait

Αυτός ο πίνακας παρουσιάζει τον αριθμό των #pragma omp for, με, και χωρίς αναμονή που υπάρχουν στην κάθε εφαρμογή. Όπως είναι γνωστό οι αλλαγές που έχουν γίνει μπορούν να βελτιώσουν αρκετά εφαρμογές που περιέχουν περιοχές διαμοιρασμού εργασίας. Ανάλογα με την ποσότητα τους, θα έχουμε και το ανάλογο ποσοστό βελτίωσης. Όπως φαίνεται στον πίνακα, εφαρμογές με περισσότερα for loop έχουν μεγαλύτερη βελτίωση. Φυσικά αυτό δεν είναι εντελώς απόλυτο αφού το κέρδος σε χρόνο εξαρτάται και τον φόρτο εργασίας μέσα στους βρόγχους και από τον αριθμό των επαναλήψεως.

5.3.1 Σύνοψη Συμπερασμάτων

Γενικά οι ατομικές εντολές λόγο των περιορισμένων σημείων που μπορούν να εφαρμοστούν δεν βρίσκονται γενικά στο προσκήνιο των προγραμματιστών. Η ευκολία και η απλότητα υλοποίησης τους, όπως και η δραματική αλλαγή σε χρόνο που επιφέρουν θα πρέπει να δημιουργούν δεύτερες σκέψεις. Η εφαρμογή τους πραγματικά δεν αφήνει ανεπηρέαστη οποιαδήποτε εφαρμογή. Οι ατομικές εντολές αν και μικρές σε εύρος και απλές στην υλοποίηση επιφέρουν δραματικές αλλαγές στον χρόνο εκτέλεσης των παράλληλων εφαρμογών. Απλούστερες στην εφαρμογή ακόμα και από τις κλειδαριές, είναι αναγκαίο και θεμιτό να ενσωματώνονται στον κώδικα χωρίς κανένα ενδοιασμό.

Κεφάλαιο 6

OMPi & Διεργασίες

Στο δεύτερο τμήμα της πτυχιακής αυτής εργασίας ασχολούμαστε με την ενσωμάτωση στον μεταγλωττιστή OMPi μίας καινούργια βιβλιοθήκης για υποστήριξη διεργασιών. Για να γίνει μια σωστή και κατανοητή περιγραφή της βιβλιοθήκης αυτής θα πρέπει να παρουσιαστεί η γενική λειτουργία του OMPi καθώς και ο τρόπο συνεργασίας της Runtime βιβλιοθήκης με τη βιβλιοθήκη νημάτων. Για να περιγράψουμε αυτή την λειτουργία θα χρησιμοποιήσουμε ένα απλό πρόγραμμα. Ο υπολογισμός του π είναι ένα απλό πρόβλημα που μπορεί εύκολα να παραλληλοποιηθεί.

6.1 Η λειτουργία του OMPi μέσω παραδείγματος

Η εφαρμογή που θα χρησιμοποιηθεί ως παράδειγμα είναι η προσέγγιση του π μέσω ενός ολοκληρώματος:

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

και προσεγγίζεται με το άθροισμα

$$\pi \approx \sum_{i=0}^{N-1} \frac{4W}{1+[(i+1/2)W]^2}$$

Ο αντίστοιχος σειριακός αλγόριθμος έχει ως ακολούθως

```
1   #include <omp.h>
2   #include <stdio.h>
3
4   double pi=0.0;
5
6   int main()
7   {
8     int i=0, N=512;
9     double W = 1.0/N;
10
11    for (i = 0; i < N; i++)
12    {
13       pi += 4 * W / (1 + ( i + 0.5 ) * ( i + 0.5 ) * W * W);
14    }
```

```
15
16      printf("pi= %f\n",pi);
17    }
```

Με ελάχιστες αλλαγές και προσθήκη μιας εντολής #pragma ... ο κώδικας μετατρέπεται εύκολα σε παράλληλο:

```
1     #include <omp.h>
2     #include <stdio.h>
3
4     double pi=0.0;
5
6     int main()
7     {
8        int i=0, N=512;
9        double W = 1.0/N, lpi=0;
10
11       #pragma omp parallel for shared(N,W) reduction(+:pi)
12       for (i = 0; i < N; i++)
13       {
14          pi += 4 * W / ( 1 + ( i + 0.5 ) * ( i + 0.5 ) * W * W);
15       }
16
17       printf("pi= %f\n",pi);
18    }
```

6.1.1 Ο παραγόμενος κώδικας

Όπως αναφέρθηκε ο OMPi είναι ένας source-to-source μεταφραστής που παράγει ένα ισοδύναμο πολυνηματικό κωδικά που με την σύνδεση (Linking) των αντίστοιχων βιβλιοθηκών είναι έτοιμος να εκτελεστεί.

Το παραγόμενο αρχείο περιέχει ένα πλήθος μεταβλητών και συναρτήσεων που διευρύνει το μέγεθος του σε σχέση με τον αρχικό κώδικα. Έτσι εδώ θα αναφερθεί αυτό το κομμάτι του κώδικα που μας ενδιαφέρει περισσότερο για να καταλάβουμε την λειτουργία των βιβλιοθηκών. Η περιγραφή θα γίνει τμηματικά για περισσότερη απλότητα. Το τμήμα κώδικα που περιγράφεται θα αναφέρεται πλησίον της περιγραφής.

```
1     int __ompi_main(int argc, char **argv)
2     {
3       int _xval = 0;
4
5       ort_initialize(&argc, &argv);
6       _xval = (int) __original_main(argc, argv);
7       ort_finalize(_xval);
8       return (_xval);
9     }
```

Ο πιο πάνω κώδικας παρουσιάζει την συνάρτηση "`main()`" του παραγόμενου προγράμματος. Με τη χρήση νημάτων αυτή η συνάρτηση είναι η πρώτη που εκτελείται. Στις διεργασίες ακολουθείται μια διαφορετική διαδικασία για λόγους που θα αναφερθούν αργότερα.

Όπως φαίνεται, στη συνάρτηση `__ompi_main()` καλούνται άλλες τρεις συναρτήσεις. Η `ort_initialize()`, η `__original_main()` που είναι η `main()` του αρχικού προγράμματος και τέλος η `ort_finalize()`.

Όλες οι συναρτήσεις που καλούνται και είναι της μορφής `ort_xxx` είναι κλήσεις στη βιβλιοθήκη, ενώ οι υπόλοιπες είναι συναρτήσεις που βρίσκονται στον παραγόμενο κώδικα.

6.1.1.1 Τροποποιήσεις στη βιβλιοθήκη Runtime

Καταρχήν η βασική λειτουργία της `ort_initialize()` είναι η αρχικοποίηση μερικών μεταβλητών και μεταβλητών περιβάλλοντος για την σωστή λειτουργία της βιβλιοθήκης. Επίσης ορίζει την δομή που περιέχει πληροφορίες για τον πατέρα (master) των νημάτων που θα δημιουργηθούν και αρχικοποιεί και την βιβλιοθήκη νημάτων καλώντας την συνάρτηση `ee_initialize()`. Το χαρακτηριστικό αυτής της δομής είναι ότι πρέπει να είναι ορατή από όλα τα νήματα. Λόγω της φύσης των νημάτων, όπου μπορούν να διαβάσουν οποιαδήποτε πληροφορία από τον πατέρα, δεν αντιμετωπίζεται πρόβλημα. Αντίθετα, οι διεργασίες μπορούν να διαμοιράζονται δεδομένα μόνο μέσω κοινής μνήμης που αρχικοποιείται ρητά πριν την δημιουργία τους. Έτσι η πρώτη αλλαγή που παρατηρείται στη βιβλιοθήκη Runtime – ort.c και εντός της `ort_share_globals()` που καλείται στο τέλος της `ort_initialize()`. Η αλλαγή αυτή αφορά τη δέσμευση κοινής μνήμης για την αποθήκευση των πληροφοριών του πατέρα.

Η επόμενη αλλαγή που έγινε στη βιβλιοθήκη είναι η δέσμευση ενός ακόμα κοινού χώρου. Ο χώρος αυτός δεσμεύεται πάλι εντός της `ort_share_globals()` και χρησιμοποιείται κατά την συνάντηση της εντολής `copyprivate` και `copyin`. Αυτές οι εντολές γενικά αντιγράφουν δεδομένα από την διεργασία/νήμα κάτοχο στα άλλα μέλη της ομάδας. Για να επιτευχθεί αυτό σε ένα διεργασιακό περιβάλλον, ένας κοινός χώρος διευθύνσεων μεγάλου μεγέθους (δεν ξέρουμε εξαρχής του μέγεθος των δεδομένων) θα πρέπει να δεσμευτεί ούτως ώστε ο κάτοχος να αντιγράψει εκεί τα δεδομένα για να μπορέσουν τα υπόλοιπα μέλη/διεργασίες να τα διαβάσουν.

Οι πιο πάνω αλλαγές θα μπορούσαν να γίνουν και εντός της `ort_share_globals()`. Όμως για να διατηρήσουμε όσο το δυνατό αμετάβλητο τον κώδικα που αφορά τα νήματα και να εντάξουμε ομαλά τις διεργασίες στη βιβλιοθήκη οι αλλαγές μεταφέρθηκαν στην `ort_share_globals()`. Η συνάρτηση αυτή καλείται μόνο κατά την παρουσία διεργασιών και είναι υπεύθυνη για την αποθήκευση των καθολικών μεταβλητών σε ένα κοινό χώρο διευθύνσεων. Οι κοινές μεταβλητές του προγράμματος κατά την ανάλυση του αρχικού κώδικα από τον μεταφραστή αποθηκεύονται σε μια λίστα. Έπειτα με την βοήθεια της συνάρτησης `ort_share_globals()` αντιγράφονται μαζί με τις υπόλοιπες κοινές μεταβλητές σε ένα κοινό χώρο μνήμης.

```
1  void ort_shmalloc(void **p, int size, int upd)
2  {
3     ee_shmalloc(p, size, upd);
4
5     if (!(*p))
6       ort_error(1, "shmalloc failed\n");
7  }
8
9  int ort_initialize(int *argc, char ***argv)
10 {
11   ...
12 #ifdef EE_TYPE_PROCESS
13     ort_share_globals()
14 #else
15   ...
16 }
```

```
1  void ort_share_globals()
2  {
3     ...
```

```
4       if (ort_sglvar_list != NULL)
5     {
6       ort_shmalloc(&ort_sglvar_area,ort_sglvar_list->size+sizeof(struct ort_struct)
7                    +sizeof(ort_eecb_t)+1000,(int) &shared_data_id);
8       memcpy(ort_sglvar_area+ort_sglvar_list->size, ort, sizeof(struct ort_struct));
9       ort=ort_sglvar_area+ort_sglvar_list->size;
10      memcpy(ort_sglvar_area+ort_sglvar_list->size+sizeof(struct ort_struct),
11             ort_master, sizeof(ort_eecb_t));
12      ort_master=ort_sglvar_area+ort_sglvar_list->size+sizeof(struct ort_struct);
13      broadcast_data = ort_sglvar_area+ort_sglvar_list->size+sizeof(struct ort_struct)
14                       +sizeof(ort_eecb_t);
15      __SETMYCB(ort_master);
16    }
17    else
18    {
19      ort_shmalloc(&ort_sglvar_area,sizeof(struct ort_struct)+ sizeof(ort_eecb_t)+1000,
20                   (int) &shared_data_id);
21      memcpy(ort_sglvar_area, ort, sizeof(struct ort_struct));
22      ort=ort_sglvar_area;
23      memcpy(ort_sglvar_area+sizeof(struct ort_struct), ort_master, sizeof(ort_eecb_t));
24      ort_master=ort_sglvar_area+sizeof(struct ort_struct);
25      broadcast_data = ort_sglvar_area+sizeof(struct ort_struct)+sizeof(ort_eecb_t);
26      __SETMYCB(ort_master);
27    }
28
29     ...
30  }
```

Να διευκρινίσουμε εδώ ότι η δημιουργία των διεργασιών γίνεται αμέσως πριν την συνάντηση μιας παράλληλης περιοχής και τερματίζονται αμέσως μετά το τέλος της. Παρατηρούμε ότι η δέσμευση της οποιασδήποτε κοινής μνήμης πραγματοποιείται πριν ακόμα δημιουργηθούν οι διεργασίες, δηλαδή πριν εκτελεστεί οποιαδήποτε παράλληλη περιοχή, έτσι δεν υπάρχει κάποιο πρόβλημα κατά την ανάγνωση του κοινού χώρου διευθύνσεων από τις διεργασίες.

6.1.1.2 Η παράλληλη περιοχή

Όπως αναφέρθηκε, στη βιβλιοθήκη Runtime, τα τμήματα των νημάτων/διεργασιών και του ελεγκτή εκτέλεσης είναι εντελώς ανεξάρτητα. Δηλαδή καμία αλλαγή δεν χρειάζεται να γίνει στον κωδικά του ort.c όταν προσαρμοστεί κάποια βιβλιοθήκη νημάτων στον OMPi. Αυτή η πολιτική είχε ακολουθηθεί και κατά προσθήκη των διεργασιών. Απλά, χρειάστηκε σε ελάχιστα σημεία να προστεθεί κώδικας ο οποίος αφορούσε μόνο διεργασίες και ο οποίος δεν εμφανίζεται στην περίπτωση των νημάτων. Αυτό έγινε δυνατό με την κατά-συνθήκη μεταφραστική λειτουργία του προεπεξεργαστή.

```
1  #ifdef EE_TYPE_PROCESS
2     ...
3  #endif
```

Προχωρώντας όμως, αυτό δεν μπόρεσε να επιτευχθεί πλήρως και έτσι προσπαθήσαμε να μειώσουμε των αριθμό των αλλαγών στο ελάχιστο, καθώς υπήρχαν σημεία στον κώδικα που έπρεπε να τροποποιηθούν. Έκτος από τις αλλαγές που έχουν αναφερθεί, εκκρεμούν ακόμη μερικές πολύ μικρές που θα αναφερθούν αργότερα.

Η ort_finalize() ως τελευταία συνάρτηση που καλείται στον παραγόμενο κώδικα προς τη βιβλιοθήκη, αποδεσμεύει όλους τους κοινούς χώρους μνήμης που έχει δεσμεύσει η ort_initialize() και η ort_share_globals() ελευθερώνοντας τους πόρους του συστήματος. Στο τέλος τερματίζει την εκτέλεση του προγράμματος.

Η συνάρτηση __original_main(argc, argv) σκόπιμα αφέθηκε τελευταία γιατί χρειάζεται περισσότερες εξηγήσεις. Η συνάρτηση αυτή περιέχει την κύρια συνάρτηση του προγράμματος. Δηλαδή μέχρι τώρα, καθόλου κώδικας από το αρχικό πρόγραμμα δεν έχει εκτελεστεί. Φυσικά ο κώδικας έχει τροποποιηθεί με βάση τις εντολές OpenMP που έχουν ενταχθεί. Έτσι ο νέος κώδικάς είναι ο ακόλουθος:

```
1  int __original_main(int _argc_ignored, char ** _argv_ignored)
2  {
3    int i = 0, N = 512;
4    double W = 1.0 / N, lpi = 0;
5
6    {
7      /* (l11) #pragma omp parallel shared(N, W) */
8      # 1 "scanner_string_buffer"
9      struct __shvt__ {
10         double (* lpi);
11         double (* W);
12         int (* N);
13        } _shvars = {
14         &lpi, &W, &N
15     };
16     ort_execute_parallel(-1, _thrFunc0_, (void *) &_shvars);
17   }
18   printf("pi= %f\n", (*pi));
19 }
```

Όπως φαίνεται ο κώδικας έχει αλλάξει αρκετά. Καταρχήν βλέπουμε ότι η εντολή #pragma omp parallel shared(N, W) έχει αντικατασταθεί με μια δομή και μια συνάρτηση, την ort_execute_parallel(). Είναι γνωστό ότι η φράση shared(N, W) κάνει κοινές, μεταξύ των νημάτων/διεργασιών τις μεταβλητές που δέχεται σαν όρισμα. Ο τρόπος με τον οποίο υλοποιείται αυτό στον OMPi είναι μέσω μιας δομής με δείκτες προς τις κοινές μεταβλητές, που δίνεται σε κάθε νήμα. Αυτή η δομή είναι η _shvars και εισάγεται σαν όρισμα στη συνάρτηση ort_execute_parallel(). Έπισης άλλο ένα όρισμα της συνάρτησης είναι η _thrFunc0_. Αυτό το όρισμα είναι μια συνάρτηση που περιέχει τον κώδικά που βρίσκεται μέσα στην πρώτη παράλληλη περιοχή. Δηλαδή τον κώδικα που θα εκτελεστεί παράλληλα από όλα τα νήματα.

Οι αρμοδιότητες της ort_execute_parallel() είναι:

1. Καθορισμός του αριθμού των νημάτων/διεργασιών που θα εκτελέσουν την παράλληλη περιοχή.
2. Αρχικοποίηση μεταβλητών για την προετοιμασία δημιουργίας των νημάτων/διεργασιών.
3. Δημιουργία της ομάδας νημάτων/διεργασιών.
4. Συμμετοχή στην ομάδα (νήμα/διεργασία πατέρας).
5. Αναμονή για τερματισμό των διεργασιών–παιδιά που δημιουργήθηκαν.

Ο καθορισμός των νημάτων/διεργασιών που θα εκτελέσουν την περιοχή εξαρτάται από την βιβλιοθήκη νημάτων/διεργασιών. Η `ort_execute_parallel()` ζητά από την βιβλιοθήκη έναν αριθμό νημάτων/διεργασιών και τις επιστρέφεται ο αριθμός που μπορεί να της δοθεί. Μετά την προετοιμασία μερικών μεταβλητών δημιουργούνται τα νήματα/διεργασίες πάλι μέσω αίτησης στην βιβλιοθήκη και τους δίνεται να εκτελέσουν την συνάρτηση `_thrFunc0_`. Την ίδια συνάρτηση εκτελεί και ο πατέρας, που λαμβάνει μέρος στην ομάδα, και μόλις τελειώσει περιμένει τα παιδιά του να τερματίσουν.

Ένα σημείο που χρειάζεται περισσότερη διευκρίνηση είναι η προετοιμασία που χρειάζονται τα νήματα/διεργασίες προτού εκτελέσουν την συνάρτηση. Η κάθε διεργασία για να μπορέσει να εκτελέσει την συνάρτηση θα πρέπει να της γνωστοποιηθούν ορισμένες μεταβλητές, όπως πληροφορίες για τον πατέρα, την συνάρτηση που θα εκτελέσουν και τις κοινές μεταβλητές. Για αυτή την εργασία υπεύθυνη είναι η `ort_get_thread_work()` που εκτελείται από κάθε νήμα/διεργασία πριν την εκτέλεση της παράλληλης περιοχής. Ουσιαστικά κάθε νήμα διαβάζει της πληροφορίες από τον πατέρα της ομάδας. Για τις διεργασίες διαβάζονται από την κοινή μνήμη που έχει δεσμευτεί αρχικά από την `ort_initialize()`.

Για κάθε παράλληλη περιοχή ακολουθείται η ίδια διαδικασία. Κάθε παράλληλη περιοχή, προφανώς θα έχει τη δική της συνάρτηση `_thrFuncX_` στον παραγόμενο κώδικα, όπως επίσης και διαφορετική `ort_execute_parallel()`.

Πιο κάτω ακολουθεί ο τροποποιημένος κώδικας της παράλληλης περιοχής. Η συναρτήσεις που μπορεί περιέχει ποικίλουν ανάλογα με τις εντολές που έχουν ανατεθεί στα νήματα/διεργασίες προς εκτέλεση. Ως μια γεύση της υλοποίησης των συναρτήσεων θα περιγραφούν με συντομία αυτές που παρουσιάζονται στην εφαρμογή υπολογισμού του π.

Στο συγκεκριμένο παράδειγμα με την προσέγγιση του π οι εντολές που χρησιμοποιήθηκαν εντός της παράλληλης περιοχής είναι οι `#pragma omp for` και η `#pragma omp atomic`. Οι επαναλήψεις στον βρόγχο διαμοιράζονται στατικά στα νήματα. Δηλαδή αν έχουμε 100 επαναλήψεις και 4 νήματα/διεργασίες κάθε ένα θα εκτελέσει 25 συνεχόμενες επαναλήψεις.

Καταρχήν στα αρχικά στάδια της συνάρτησης αρχικοποιούνται οι δείκτες μιας δομής να δείχνουν στις κοινές μεταβλητές της παράλληλης περιοχής. Παρατηρήστε ότι οι μεταβλητές λαμβάνονται από τη συνάρτηση `ort_ get_shared_vars()` με όρισμα την δομή `__me`. Η δομή αυτή περιέχει πληροφορίες για το τρέχων νήμα καθώς και την διεύθυνσή του πατέρα όπου εκεί βρίσκονται αποθηκευμένες οι κοινές μεταβλητές. Αυτή είναι η δομή που αρχικοποιείται από την `ort_get_thread_work()`. Μετά την αρχικοποίηση των κοινών μεταβλητών ακολουθεί η εκτέλεση του βρόγχου.

Πρώτο βήμα για την εκτέλεση του βρόγχου είναι το όρισμα κάποιων μεταβλητών (π.χ για αποθήκευσή του βήματος επανάληψης) και ακολουθεί η εκτέλεση της συνάρτηση `ort_entering_for()`. Αυτή η συνάρτηση δημιουργεί μια νέα περιοχή διαμοιρασμού εργασία τύπου `for` και την αρχικοποιεί. Αυτό γίνεται από το νήμα που θα προσπελάσει πρώτο την συνάρτηση.

Υπεύθυνη συνάρτηση για διαμοιρασμό των επαναλήψεων στα νήματα/διεργασίες είναι η `ort_get_ static_default_chunk()`. Η συνάρτηση αρχικοποιεί τις μεταβλητές `step_, &from_, &to_` με τις ανάλογες τιμές για να εκτελεστεί ο βρόγχος με την σωστή ακολουθία επαναλήψεων.

Οι εντολές `ort_reduction_begin()` και `ort_reduction_end()` που ακολουθούν, ασφαλίζουν και ελευθερώνουν μια κλειδαριά αντίστοιχα για να διαφυλάξουν το άθροισμα πάνω στην καθολική μεταβλητή `pi`.

Ο κώδικας που ακολουθεί παρουσιάζει όσα αναφέραμε πιο πάνω:

```
1  static void * _thrFunc0_(void * __me)
2  {
3    struct __shvt__ {
4        double (* pi);
5        double (* W);
6        int (* N);
7      };
8    struct __shvt__ * _shvars =
9                  (struct __shvt__ *) ort_get_shared_vars(__me);
10   double (* W) = _shvars->W;
```

```
11    int (* N) = _shvars->N;
12    double pi = 0;
13
14    /* (l11) #pragma omp parallel shared(N, W)
15             reduction(+ : pi) -- body moved below */
16    {
17      /* #pragma omp for  */
18      int i;
19      int from_ = 0, to_ = 0, step_;
20      struct _ort_gdopt_ gdopt_;
21
22      step_ = 1;
23      ort_entering_for(1, 0, 0, step_, &gdopt_);
24      if (ort_get_static_default_chunk
25                                (0, (*N), step_, &from_, &to_))
26      {
27        for (i = from_; i < to_; i = i + 1)
28        {
29          pi += 4 * (*W) / (1 + (i + 0.5) * (i + 0.5) * (*W) * (*W));
30        }
31      }
32      ort_leaving_for();
33    }
34    ort_reduction_begin(&_paredlock0);
35    *(_shvars->pi) += pi;
36    ort_reduction_end(&_paredlock0);
37    return (void *) 0;
38  }
```

Μετά το τέλος του πιο πάνω κώδικα, τα νήματα/διεργασίες τερματίζουν (`ort_ finalize()`) και ο πατέρας (mater) συνεχίζει την σειριακή εκτέλεση του κώδικα με την εκτύπωση του αποτελέσματος.

Έχοντας τώρα σχηματίσει μια βασική εικόνα του τρόπου «συνεργασίας» παραγόμενου κώδικα με τις δύο βιβλιοθήκες Runtime και νημάτων/διεργασιών μπορούμε να προχωρήσουμε σε μια πιο βαθιά περιγραφή της βιβλιοθήκης διεργασιών ee_process και συγκεκριμένα του αρχείου oprc.c .

6.2 Βιβλιοθήκη Διεργασιών

Καταρχήν προτού περιγράψουμε τον κώδικα της βιβλιοθήκης ας δούμε τις κύριες διαφορές σε σχέση με τη βιβλιοθήκη νημάτων. Ο OMPi στο προηγούμενο παράδειγμα υπολογισμού του π ξεκίνησε την εκτέλεση του κώδικα απευθείας από τον παραγόμενο κώδικα. Με τις διεργασίες όμως η εκτέλεση του κώδικα αρχίζει μέσα από την αρχικοποίηση της βιβλιοθήκη διεργασιών. Ο λόγος που γίνεται αυτό αφορά την κοινή μνήμη και την προσπέλασή της από τις διεργασίες. Ας πάρουμε τα πράγματα από την αρχή.

6.2.1 Αρχιτεκτονική βιβλιοθήκης Παραγωγού – Καταναλώτη

Η εκτέλεση του παραγόμενου κώδικα θα ανατεθεί σε δύο οντότητες, που θα ακολουθούν την αρχιτεκτονική του Παραγωγού – Καταναλωτή (Producer – Consumer):

- Την αρχική διεργασία που θα αναλάβει την δημιουργία (παραγωγή) διεργασιών κατόπιν αιτήσεως από την Runtime βιβλιοθήκη.

- και ένα νήμα(καταναλωτή) που θα εκτελεί τον όλο κώδικα και θα συμμετάσχει στον κώδικα εκτέλεσης της ομαδας. (νήμα master)

Όπως αναφέραμε, οι διεργασίες παιδιά πρέπει με κάποιο τρόπο να διαβάζουν πληροφορίες από πατέρα. Όμως οι διεργασίες μόνο μέσω κοινής μνήμη που δεσμεύεται πριν την δημιουργία τους μπορούν να διαμοιράζονται πληροφορίες. Έτσι κατά την αρχικοποίηση της βιβλιοθήκης δεσμεύεται ένας χώρος κοινής μνήμης και αποθηκεύεται εκεί η στοίβα του πατέρα. Δηλαδή του νήματος που θα εκτελέσει την συνάρτηση `__ompi_main()`. Με τον τρόπο αυτό οι διεργασίες παιδιά έχοντας στην κοινή μνήμη την στοίβα του πατέρα, μπορούν να διαβάσουν οποιαδήποτε πληροφορία από αυτόν. Αντίθετα, στα νήματα αυτό δεν ήταν απαραίτητο, αφού εξαρχής, αμέσως μετά την δημιουργία τους μπορούν να διαβάσουν μεταβλητές από τον πατέρα. Ακολούθως η κύρια διεργασία που δημιουργεί το νήμα πατέρα αφήνεται στο να παράγει τις διεργασίες κατόπιν αίτησης από την Runtime βιβλιοθήκη. Ο κώδικας της βιβλιοθήκης διεργασιών θα παρουσιαστεί τμηματικά πιο κάτω.

```
1  void oprc_shmalloc(void **p ,size_t size,int *memid)
2  {
3     ...
4     shmget(IPC_PRIVATE,size,0600|IPC_CREAT);
5     ...
6     shmat(*memid, 0, 0);
7     ...
8  }
9  void oprc__ompi_main(void)
10 {
11    __ompi_main(ort_argc,ort_argv);
12 }
13
14 int main(int argc, char **argv)
15 {
16    ...
17    oprc_shmalloc(&seg_addr,(size_t)1024*1024,(int)&memid);
18    ...
19    pthread_attr_init(&tattr);
20    pthread_attr_setstack(&tattr,p,1024*1024);
21    ...
22    pthread_create(&tid,&tattr,(void *)oprc__ompi_main,NULL);
23    ...
24 }
```

Όπως φαίνεται και στον κώδικα προτού δημιουργηθεί το νήμα πατέρα, η στοίβα του τοποθετείται στο κοινό χώρο που έχει δεσμευτεί και έπειτα εκτελεί μια τοπική συνάρτηση την `oprc_ompi_main()`. Η συνάρτηση αυτή με τη σειρά της εκτελεί την `__ompi_main()`. Η `__ompi_main()` εκτελείται με αυτό τον έμμεσο τρόπο για να μπορέσουν οι παράμετροι `argc, argv` να περαστούν στη συνάρτηση. Αυτό γίνεται γιατί κατά την εκτέλεση μιας συνάρτηση από ένα νήμα δεν μπορούν να εισαχθούν απευθείας ορίσματα.

Προηγουμένως αναφέρθηκε ότι η γονική διεργασία (αυτή που δημιουργεί το νήμα) αφήνεται στο να δημιουργεί της διεργασίες κατόπιν αιτήσεως της βιβλιοθήκης `Runtime`. Αυτό το αίτημα, γίνεται μέσο της συνάρτησης `oprc_create()`. Για να υπάρξει σωστή συνεργασία μεταξύ της γονικής διεργασίας και του νήματος θα πρέπει να υπάρξει κάποιου είδους συντονισμού. Δηλαδή όταν γίνει μια αίτηση, το νήμα πρέπει να περιμένει την δημιουργία των διεργασιών προτού προχωρήσει στην εκτέλεση της παράλληλης περιοχής. Αυτός ο συντονισμός γίνεται μέσο μεταβλητών συνθηκών (`condition variables`). Όπως παρουσιάζεται στον κώδικα, η γονική διεργασία εκτελεί ένα ατέρμονο βρόχο

while(!end). Κατά την εκτέλεση του βρόγχου, κοιμάται, και περιμένει για να εκτελέσει την συνάρτηση pfork() κατόπιν αφύπνισης της από το νήμα. Η συνάρτηση pfork() είναι μια τροποποίηση της συνάρτησης fork() και επιστέφει τον αύξων αριθμό της διεργασίας που έχει δημιουργηθεί. Μετά την αφύπνιση της γονικής διεργασίας το νήμα, τίθεται σε αδράνεια μέχρι να δημιουργηθούν οι διεργασίες. Όταν η γονική διεργασία ολοκληρώσει την εργασία της, ξυπνά με ένα σήμα το νήμα και συνεχίζεται κανονικά η εκτέλεση, ενώ αυτή επανεκτελεί τον βρόγχο και αδρανοποιείται ξανά.

```
1  void oprc_create(int numthr, int level,
2                   void *(*workfunc)(void*), void *arg, void **ignore)
3  {
4    if (numthr <= 0) return;
5
6    number_of_process = numthr;
7    team_workfunc = workfunc;
8    team_argument = arg;
9
10   pthread_mutex_lock(&wait_thread_mutex);
11   pthread_mutex_lock(&wait_process_mutex);
12   pthread_cond_signal(&wait_thread);
13   pthread_mutex_unlock(&wait_thread_mutex);
14
15   pthread_cond_wait(&wait_process,&wait_process_mutex);
16   pthread_mutex_unlock(&wait_process_mutex);
17 }
18
19 int main(int argc, char **argv)
20 {
21     ...
22     while(!end)
23     {
24       pthread_cond_wait(&wait_thread,&wait_thread_mutex);
25       pthread_mutex_unlock(&wait_thread_mutex);
26       FENCE;
27
28       if(!end)
29       {
30          pid=pfork(number_of_process);
31
32          if(pid!=0)
33          {
34            /* We have to do the following, before the actual execution */
35            ort_get_thread_work(pid, team_argument, NULL, &arg);
36
37            (*team_workfunc)(arg); /* Execute requested code */
38
39            exit(0);
40          }
41       }
42       pthread_mutex_lock(&wait_process_mutex);
43       pthread_mutex_lock(&wait_thread_mutex);
44       pthread_cond_signal(&wait_process);
45       pthread_mutex_unlock(&wait_process_mutex);
```

```
46      }
47      ...
48  }
```

Τέλος για να τερματιστεί μια εφαρμογή, εκτός από την κλήση `ort_finalize` εκτελείται και η συνάρτηση `oprc_finalize` που βρίσκεται εντός της βιβλιοθήκης διεργασιών. Στη συνάρτηση αυτή τίθεται η μεταβλητή `end = 0` για να μην εκτελεστεί ξανά ο βρόγχος για τη δημιουργία διεργασιών. Όμως η διεργασία γονέας περιμένει είδη μέσα στο βρόγχο. Έτσι χρειάζεται αφύπνιση. Και αυτό γίνεται από την `ort_finalize`. Επίσης ο επιπλέον έλεγχος που γίνεται πριν την εκτέλεση της `create_process()` επιβάλλεται για να αποτραπεί η δημιουργία νέων διεργασιών κατά το τέλος εκτέλεσης της εφαρμογής. Έτσι ο κώδικας τερματίζει με την απελευθέρωση της κοινής μνήμης των `mutexs` και των μεταβλητών συνθηκών.

```
1  int main(int argc, char **argv)
2  {
3      ...
4      oprc_shmfree(&memid);
5      oprc_shmfree(&ort_lock_id);
6      oprc_shmfree(&ort_locks_id);
7
8      pthread_attr_destroy(&tattr);
9      pthread_mutex_destroy(&wait_thread_mutex);
10     pthread_mutex_destroy(&wait_process_mutex);
11
12     pthread_cond_destroy(&wait_thread);
13     pthread_cond_destroy(&wait_process);
14
15     exit(0);
16 }
```

6.2.1.1 Υλοποίηση Κλειδαριών

Ένα άλλο πρόβλημα που χρειάζεται τροποποίηση στη βιβλιοθήκη διεργασιών, είναι η υλοποίηση των κλειδαριών. Κάθε κλειδαριά κατά την αρχικοποίηση της δεσμεύει χώρο και ακολούθως αποθηκεύεται εκεί. Στη βιβλιοθήκη νημάτων η δέσμευση μνήμης `malloc()` δεσμεύει χώρο στην στοίβα της διεργασίας. Στις διεργασίες δεν μπορούμε να δεσμεύουμε κοινό χώρο μνήμης αφού δημιουργηθούν οι διεργασίες γιατί δεν θα μπορούν αν τον διαβάσουν. Έτσι πρέπει να αναζητήσουμε λύση με άλλους τρόπους.

Η λύση που εφαρμόστηκε σε αυτό το πρόβλημα είναι κάπως περιοριστική. Κατά την αρχικοποίηση της βιβλιοθήκης διεργασιών με την εκτέλεση της συνάρτηση `oprc_initialize()` δεσμεύεται ένας κοινός χώρος μνήμης μεγέθους 'N' κλειδαριών. Σε αυτό τον κοινό χώρο μνήμης, που μπορούν να προσπελάσουν όλες οι διεργασίες, αρχικοποιούνται ακολουθιακά οι κλειδαριές που θα χρειαστεί η εφαρμογή (π.χ κλειδαριές χρήστη). Κατά την αρχικοποίηση μιας νέας κλειδαριάς `oprc_init_lock()` πραγματοποιείται ένας έλεγχος με συνθήκη `if`. Αυτός ο έλεγχος γίνεται για να καθοριστεί αν η κλειδαριά είναι κλειδαριά χρήστη (δηλαδή πρέπει να αποθηκευτεί στην κοινή μνήμη) ή υπάρχει είδη δεσμευμένος χώρος (κοινός προς όλους) στον πατέρα. Αν είναι τύπου χρήστη αποθηκεύεται στη ήδη δεσμευμένη μνήμη. Μια μεταβλητή `(number_of_lock)` που αυξάνεται κατά ένα δείχνει την θέση της νέας κλειδαριάς στην κοινή μνήμη. Επίσης η μεταβλητή λαμβάνεται από το υπόλοιπο της διαίρεσης με τον αριθμό των κλειδαριών 'N' για να μην μπορεί να τον ξεπεράσει. Ο περιορισμός που υπάρχει με αυτή την λύση είναι ότι δεν μπορούν να δεσμευτούν 'N' κλειδαριές ταυτόχρονα στο κώδικα. Αν και ο αριθμός 'N' είναι μεγάλος το πρόβλημα εξακολουθεί να υφίσταται, αφού με τον υπάρχον μεταφραστή δεν μπορούμε να γνωρίζουμε εκ των προτέρων των αριθμό των κλειδαριών που υπάρχουν στον κώδικα.

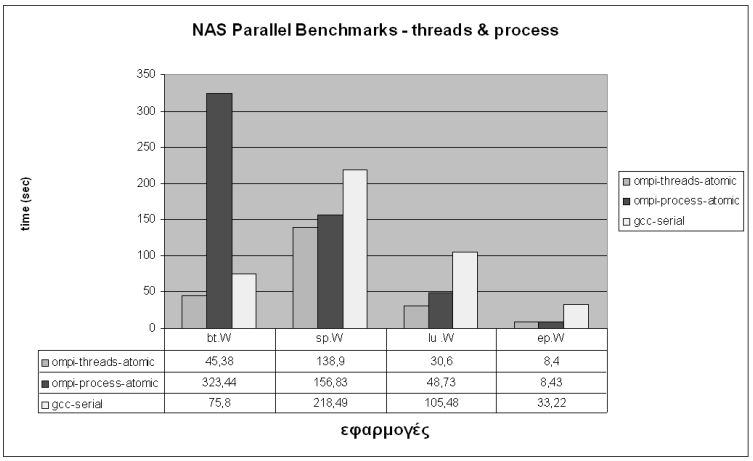
```
int oprc_initialize(int *argc, char ***argv, ort_icvs_t *icv,
                    ort_caps_t *caps)
{
   ...
  oprc_shmalloc(&number_of_lock,(size_t)sizeof(int),
                (int) &ort_lock_id);
  oprc_shmalloc(&shlock,(size_t)100*sizeof(oprc_lock_t),
                (int) &ort_locks_id);
  *number_of_lock=0;
   ...
}

int oprc_init_lock(oprc_lock_t *lock, int type)
{
  if(lock == -1)
  {
      lock = shlock + (*number_of_lock);
      *number_of_lock=((*number_of_lock)+1)%100;
      FENCE;
  }
  ...
  return(lock);
}
```

Επίσης δύο μικρές τροποποιήσεις που έγιναν στη βιβλιοθήκη Runtime και δεν αναφέρθηκαν πριν αφορούν τις κλειδαριές. Όπως είπαμε οι κλειδαριές στα νήματα δεσμεύονται στη στοίβα της διεργασίας. Αφού χρησιμοποιηθούν και δεν είναι χρήσιμες πλέον, η μνήμη που δεσμεύουν ελευθερώνεται από την συνάρτηση `omp_destroy_lock()`. Αυτό δεν μπορεί να γίνει για την μνήμη που δεσμεύεται για τις διεργασίες. Η μνήμη αυτή πρέπει να αποδεσμευτεί στο τέλος της εφαρμογής δηλαδή κατά την κλήση της συνάρτησης `oprc_finalize`.

Επίσης στη Runtime βιβλιοθήκη κατά τη κλήση της συνάρτηση `ort_prepare_omp_ lock()` καθορίζεται ο τύπος της κλειδαριάς για να μπορεί η βιβλιοθήκη διεργασιών να δεσμεύσει τον κατάλληλο χώρο. Να διευκρινίσουμε εδώ ότι η συνάρτηση `oprc_init_ lock()` επιστέφει την διεύθυνση στον κοινό χώρο μνήμης της νέας κλειδαριάς.

6.3 NAS Parallel Benchmarks & Διεργασίες

Για να συγκρίνουμε την απόδοση των διεργασιών σε σχέση με τα νήματα εκτελέσαμε μερικές από τις εφαρμογές των NAS Parallel Benchmarks και πήραμε συγκριτικά αποτελέσματα. Αδιαμφισβήτητα έχουμε καλύτερους χρόνους στα νήματα παρά στις διεργασίες. Η γραφική παράσταση που ακολουθεί παρουσιάζει τις αντίστοιχες συγκριτικές μετρήσεις.



Σχήμα 6.1: Αποτελέσματα – NAS Threads & Process

Όπως φαίνεται και από τον πιο πάνω πίνακα, τα νήματα υπερέχουν κατά πολύ σε σχέση με τις διεργασίες. Στην εφαρμογή BT βλέπουμε μια διαφορά γύρο στα 280 δευτερόλεπτα. Τα νήματα χρειάστηκαν μόλις 45 δευτερόλεπτα για να τρέξουν την εφαρμογή ενώ οι διεργασίες 325. Οι άλλες δύο εφαρμογές παρουσιάζουν μικρότερη διαφορά στο χρόνο εκτέλεσης γύρω στα 20 δευτερόλεπτα ενώ η τελευταία είναι αμετάβλητη.

Ο σειριακός χρόνος εκτέλεσης φαίνεται να είναι αρκετά μεγαλύτερος από τον παράλληλο με νήματα και διεργασίες, εξαίρεση την εφαρμογή BT. Όπως παρατηρείται σε μερικές περιπτώσεις ο σειριακός χρόνος είναι καλύτερος ακόμα και από τον παράλληλο. Αυτό εξαρτάτε από τη φύση της εφαρμογής και τον τρόπο που γίνεται ο παράλληλος προγραμματισμός.

Βιβλιογραφία

[1] V. V. Dimakopoulos, E. Leontiadis and G. Tzoumas, A Portable C Compiler for OpenMP V.2.0, In *Proc. of the 5th Europian Workshop on OpenMP (EWOMP '03)*, Aachen, Germany, October 2003.

[2] P. E. Hadjidoukas and V. V Dimakopoulos, Nested Parallelism in the OMPi OpenMP C Compiler, In *Proc. of the European Conference on Parallel Computing (EUROPAR '07)*, Rennes, France, August 2007.

[3] J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, In *Proc. of the 1st EWOMP, European Workshop on OpenMP*, Lund, Sweden, 1999.

[4] J. M. Bull and D. O'Neill, A Microbenchmark Suite for OpenMP 2.0, In *Proc. of the 3th EWOMP, European Workshop on OpenMP*, Barcelona, Spain, 2001.

[5] OpenMP Architecture Review Board: OpenMP and C++ Application Pogram Interface, Version 2.5, May 2005.

[6] H. Jin, M. Frumkin, and J. Yan, The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance, Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[7] G. C. Philos, V. V. Dimakopoulos and P. E. Hadjidoukas, A runtime architecture for ubiquitous support of OpenMP, In *Proc of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*, Krakow, Poland, July 2008, to appear.

[8] V. V. Dimakopoulos, P. E. Hadjidoukas and G. C. Philos, A Microbenchmark Study of OpenMP Overheads Under Nested Parallelism, In *Proc. of the 4th International Workshop on OpenMP (IWOMP'08)*, West Lafayette, IN, USA, May 2008, 1–12.

[9] Netscape Portable Runtime Reference Manual
http://www.mozilla.org/projects/nspr/reference/html/index.html

[10] GLib Reference Manual
http://library.gnome.org/devel/glib/

[11] HP: The atomic_ops project
http://www.hpl.hp.com/research/linux/atomic_ops/

[12] Intel Architecture Software Developer's Manual
Volume 2: Instruction Set Reference

Κεφάλαιο 7

Παραρτήματα

Α΄ Παράρτημα Εγκατάσταση του OMPi

Τον κώδικα του μεταγλωττιστή OMPi μπορείται να τον προμηθευτείται από την επίσημη ιστοσελίδα του

`http://www.cs.uoi.gr/ ompi/`

Η διαδικασία εγκατάστασης είναι γρήγορη και απλή.

Πρώτα αποσυμπιέζεται το αρχείο με τις εντολές

```
gunzip ompi-X.Y.Z.tar.gz
tar xvf ompi-X.Y.Z.tar
```

Τρέξτε το script configure με το επιθυμητό `<install-dir>`

```
./configure --prefix=<install-dir>
```

Ακολούθως μεταφράστε και εγκαταστήστε το πακέτο

```
make
make install
```

Φροντίστε η ο φάκελος `<install-dir>/bin` να βρίσκεται στη μεταβλητή περιβάλλοντός σας `PATH`.

Για αλλαγή βιβλιοθήκης νημάτων/διεργασιών μπορείται κατά την εκτέλεση του script configure να θέσετε την παράμετρο `--with-ortlib=<name> ...` δηλαδή ως εξής:

```
./configure --with-ortlib=<name> ...
```

με name μιά από τις υπάρχουσες βιβλιοθήκες. Για διεργασίες γράφεται `process`. Με την παράμετρο αυτή, καθορίζεται πια βιβλιοθήκη νημάτων/διεργασιών θα συνδεθεί (Linking) μαζί με τον παραγόμενο κώδικα και τη βιβλιοθήκη Runtime για να δημιουργηθεί το εκτελέσιμο αρχείο.
Αφού πραγματοποιηθεί εγκατάσταση του OMPi, μπορούμε απλά να του δώσουμε σαν είσοδο ένα αρχείο C με OpenMP εντολές και να μας παράγει το εκτελέσιμο `a.out` με την εντολή:

```
ompicc [-k] code.c
```

Η παράμετρος [-k] παράγει και ένα αρχείο με όνομα code_ompi.c με τον κώδικα που παράγει ο OMPi.

Β′ Παράρτημα
Ο κώδικας της βιβλιοθήκης διεργασιών

```
1  /*
2    OMPi OpenMP Compiler
3    == Copyright since 2001 the OMPi Team
4    == Department of Computer Science, University of Ioannina
5
6    This file is part of OMPi.
7
8    OMPi is free software; you can redistribute it and/or modify
9    it under the terms of the GNU General Public License as published by
10   the Free Software Foundation; either version 2 of the License, or
11   (at your option) any later version.
12
13   OMPi is distributed in the hope that it will be useful,
14   but WITHOUT ANY WARRANTY; without even the implied warranty of
15   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16   GNU General Public License for more details.
17
18   You should have received a copy of the GNU General Public License
19   along with OMPi; if not, write to the Free Software
20   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
21 */
22
23 #include <pthread.h>
24 #include <stdio.h>
25 #include <unistd.h>
26 #include <stdlib.h>
27 #include <unistd.h>
28 #include <sys/types.h>
29 #include <sys/ipc.h>
30 #include <sys/shm.h>
31 #include <sys/wait.h>
32 #include <sys/shm.h>
33 #include "../ort.h"
34 #include "ee.h"
35 #define WAIT_WHILE(f) { FENCE; for(;f;) oprc_yield(); }
36
37 static int pthread_lib_inited = 0;     /* Flag to avoid re-initializations */
38
39 /* Global stuff for the team */
40 static void *team_argument;            /* The argument each thread must init */
41 static void *(*team_workfunc)(void *); /* What function to execute */
42
43 pid_t *seg_addr;
44 volatile int end=0;
45
46 pthread_mutex_t wait_thread_mutex,wait_process_mutex;
47 pthread_cond_t wait_thread, wait_process;
48
49 int *number_of_lock=NULL;
50 int ort_lock_id=-1;
51 int ort_locks_id=-1;
52 oprc_lock_t *shlock=NULL;
53
54 int number_of_process;
```

```
55 int ort_argc;
56 char **ort_argv;
57
58 int oprc_pid(){return(0);}
59
60 int pfork(int N)
61 {
62    int i=0;
63
64    for(; N>0; N--)
65    {
66       if(fork()==0)   /* CHILD */
67         {
68           return(N);
69         }
70    }
71
72    return(0);         /* PARENT */
73 }
74
75 void oprc_shmfree(int *p)
76 {
77    if(shmctl(*p, IPC_RMID, 0)== -1)
78    {
79       printf("Shared Memory Free error\n");
80       exit(0);
81    }
82 }
83
84 void oprc_shmalloc(void **p ,size_t size,int *memid)
85 {
86    *memid = shmget(IPC_PRIVATE,size,0600|IPC_CREAT);
87    if(*memid == -1 )
88    {
89       printf("Shared Memory allocation error\n");
90       exit(0);
91    }
92    *p = shmat(*memid, 0, 0);
93    if(p == -1 )
94    {
95       printf("Shared Memory attach error\n");
96       exit(0);
97    }
98 }
99
100
101 /***************************
102  *                         *
103  *  Library initialization. *
104  *                         *
105  ***************************/
106 int oprc_initialize(int *argc, char ***argv, ort_icvs_t *icv, ort_caps_t *caps)
107 {
108   pid_t id;
109   int         nthr;
110
111   oprc_shmalloc(&number_of_lock,(size_t)sizeof(int),(int) &ort_lock_id);
112   oprc_shmalloc(&shlock,(size_t)100*sizeof(oprc_lock_t),(int) &ort_locks_id);
```

```
113   *number_of_lock=0;
114
115
116   nthr = (icv->nthreads > 0) ?  /* Explicitely requested population */
117               icv->nthreads :
118               icv->ncpus;      /* Use a pool of #cpus threads otherwise */
119   caps->supports_nested            = 0;
120   caps->supports_dynamic           = 1;
121   caps->supports_nested_nondynamic = 0;
122   caps->max_levels_supported       = 1;
123   caps->default_numthreads         = nthr;
124   caps->max_threads_supported      = -1;     /* No limit */
125
126   if ( pthread_lib_inited ) return (0);
127
128   pthread_lib_inited = 1;
129   return (0);
130 }
131
132 void oprc_finalize(int exitvalue)
133 {
134   end=1;
135   FENCE;
136
137   pthread_mutex_lock(&wait_thread_mutex);
138   pthread_cond_signal(&wait_thread);
139   pthread_mutex_unlock(&wait_thread_mutex);
140 }
141
142 /* Request for "numthr" threads to execute parallelism in nest
143  * level "level".
144  * We only support 1 level of parallelism, so we always return 0
145  * if level > 1
146  */
147 int oprc_request(int numthr, int level)
148 {
149   return ( (level == 1) ? numthr : 0 );
150 }
151
152
153 /* Only the master thread can call this.
154  * It blocks the thread waiting for all its children to finish their job.
155  */
156 void oprc_waitall(void **ignore)
157 {
158   int i;
159   for (i = 0; i < number_of_process; i++)
160       wait(0);
161 }
162
163
164 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
165  *                                                                     *
166  *  LOCKS                                                              *
167  *                                                                     *
168  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
169
170 int oprc_init_lock(oprc_lock_t *lock, int type)
```

```
171 {
172   if(lock == -1)
173   {
174   lock = (shlock + (*number_of_lock));
175   *number_of_lock=((*number_of_lock)+1)%100;
176   FENCE;
177   }
178
179   switch (lock->lock.type = type)
180   {
181     case ORT_LOCK_NEST:
182     {
183       oprc_nestlock_t *l = &(lock->lock.data.nest);
184
185       pthread_mutex_init(&l->ilock, NULL);
186       pthread_mutex_init(&l->lock, NULL);
187       l->count = 0;
188       pthread_cond_init(&l->cond, 0);
189       //return (0);
190       return(lock);
191     }
192
193     case ORT_LOCK_SPIN:
194     {
195 #ifdef HAVE_SPINLOCKS
196       //return pthread_spin_init(&(lock->lock.data.spin), 0);
197       pthread_spin_init(&(lock->lock.data.spin), 0);
198       return(lock);
199 #else
200   #ifdef PTHREAD_MUTEX_SPINBLOCK_NP                          /* e.g. IRIX */
201       static pthread_mutexattr_t spinblock_attr;
202       static int firstCall = 1;
203
204       if (firstCall) {
205         firstCall = 0;
206         pthread_mutexattr_init(&spinblock_attr);
207         pthread_mutexattr_settype(&spinblock_attr, PTHREAD_MUTEX_SPINBLOCK_NP);
208       }
209       lock->lock.data.spin.rndelay = 0;
210       //return pthread_mutex_init(&(lock->lock.data.spin.mutex), &spinblock_attr);
211       pthread_mutex_init(&(lock->lock.data.spin.mutex), &spinblock_attr);
212       return(lock);
213   #else
214       lock->lock.data.spin.rndelay = 0;
215       //return pthread_mutex_init(&(lock->lock.data.spin.mutex), NULL);
216       pthread_mutex_init(&(lock->lock.data.spin.mutex), NULL);
217       return(lock);
218   #endif
219 #endif
220     }
221
222     default: /* ORT_LOCK_NORMAL */
223       {
224        //return pthread_mutex_init(&(lock->lock.data.normal),NULL);
225        pthread_mutex_init(&(lock->lock.data.normal),NULL);
226        return(lock);
227       }
228   }
```

```
229 }
230
231
232 int oprc_destroy_lock(oprc_lock_t *lock)
233 {
234   switch (lock->lock.type)
235   {
236     case ORT_LOCK_NEST:
237     {
238       oprc_nestlock_t *l = &(lock->lock.data.nest);
239
240       pthread_mutex_destroy(&l->lock);
241       pthread_cond_destroy(&l->cond);
242       pthread_mutex_destroy(&l->ilock);
243       return 0;
244     }
245
246     case ORT_LOCK_SPIN:
247 #ifdef HAVE_SPINLOCKS
248       return pthread_spin_destroy(&(lock->lock.data.spin));
249 #else
250       return pthread_mutex_destroy(&(lock->lock.data.spin.mutex));
251 #endif
252
253     default: /* ORT_LOCK_NORMAL */
254        return pthread_mutex_destroy(&(lock->lock.data.normal));
255
256   }
257 }
258
259
260 int oprc_set_lock(oprc_lock_t *lock)
261 {
262   switch (lock->lock.type)
263   {
264     case ORT_LOCK_NEST:
265     {
266       oprc_nestlock_t *l = &(lock->lock.data.nest);
267
268       pthread_mutex_lock(&l->ilock);
269         if (pthread_mutex_trylock(&l->lock) == 0) /* If not locked, lock it */
270         {
271           l->owner = pthread_self();              /* Get wownership */
272           l->count++;
273         }
274         else
275           if (pthread_equal(l->owner, pthread_self()))  /* Did i do it? */
276             l->count++;
277           else                                    /* Locked by someone else */
278           {
279             while ( pthread_mutex_trylock(&l->lock) )
280               pthread_cond_wait(&l->cond, &l->ilock);
281             l->owner = pthread_self();
282             l->count++;
283           }
284       pthread_mutex_unlock(&l->ilock);
285       return (0);
286     }
```

```
287
288     case ORT_LOCK_SPIN:
289     {
290 #ifdef HAVE_SPINLOCKS
291       return pthread_spin_lock(&(lock->lock.data.spin));
292 #else
293   #ifdef PTHREAD_MUTEX_SPINBLOCK_NP                              /* e.g. IRIX */
294       return pthread_mutex_lock(&lock->data.mutex);
295   #else
296       /* General, portable solution: spin trying with exponential backoff */
297       volatile int count, delay, dummy;
298       for (delay = lock->lock.data.spin.rndelay;
299                    pthread_mutex_trylock(&(lock->lock.data.spin.mutex)); )
300       {
301         for (count = dummy = 0; count < delay; count++ )
302           dummy += count;      /* To avoid compiler optimizations */
303         if (delay == 0)
304           delay = 1;
305         else
306           if (delay < 10000)     /* Don't delay too much */
307             delay = delay << 1;
308       }
309       lock->lock.data.spin.rndelay++;    /* Next thread would wait a bit more */
310       return (0);
311   #endif
312 #endif
313     }
314
315     default: /* ORT_LOCK_NORMAL */
316       {
317         return pthread_mutex_lock(&(lock->lock.data.normal));
318       }
319   }
320 }
321
322
323 int oprc_unset_lock(oprc_lock_t *lock)
324 {
325   switch (lock->lock.type)
326   {
327     case ORT_LOCK_NEST:
328     {
329       oprc_nestlock_t *l = &(lock->lock.data.nest);
330
331       pthread_mutex_lock(&l->ilock);
332         if (pthread_equal(l->owner, pthread_self()) && l->count > 0)
333         {
334           l->count--;
335           if (l->count == 0)
336           {
337             pthread_mutex_unlock(&l->lock);
338             pthread_cond_signal(&l->cond);
339           }
340         }
341       pthread_mutex_unlock(&l->ilock);
342       return 0;
343     }
344
```

```
345     case ORT_LOCK_SPIN:
346 #ifdef HAVE_SPINLOCKS
347       return pthread_spin_unlock(&(lock->lock.data.spin));
348 #else
349       lock->lock.data.spin.rndelay = 0;   /* reset it */
350       return pthread_mutex_unlock(&(lock->lock.data.spin.mutex));
351 #endif
352 
353     default: /* ORT_LOCK_NORMAL */
354       {
355         return pthread_mutex_unlock(&(lock->lock.data.normal));
356       }
357   }
358 }
359 
360 
361 int oprc_test_lock(oprc_lock_t *lock)
362 {
363   switch (lock->lock.type)
364   {
365     case ORT_LOCK_NEST:
366     {
367       oprc_nestlock_t *l = &(lock->lock.data.nest);
368       int res;
369 
370       pthread_mutex_lock(&l->ilock);
371         if (pthread_mutex_trylock(&l->lock) == 0)
372         {
373           l->owner = pthread_self();
374           res = ++l->count;
375         }
376         else
377           if (pthread_equal(l->owner, pthread_self()))
378             res = ++l->count;
379           else
380             res = 0;
381       pthread_mutex_unlock(&l->ilock);
382       return res;
383     }
384 
385     case ORT_LOCK_SPIN:
386 #ifdef HAVE_SPINLOCKS
387       return pthread_spin_trylock(&(lock->lock.data.spin));
388 #else
389       return ( !pthread_mutex_trylock(&(lock->lock.data.spin.mutex)) );
390 #endif
391 
392     default: /* ORT_LOCK_NORMAL */
393     {
394         return(!pthread_mutex_trylock(&(lock->lock.data.normal)));
395     }
396   }
397 }
398 
399 void oprc_create(int numthr, int level,
400                  void *(*workfunc)(void*), void *arg, void **ignore)
401 {
402   if (numthr <= 0) return;
```

```
403
404   number_of_process = numthr;
405   team_workfunc = workfunc;
406   team_argument = arg;
407
408   pthread_mutex_lock(&wait_thread_mutex);
409   pthread_mutex_lock(&wait_process_mutex);
410   pthread_cond_signal(&wait_thread);
411   pthread_mutex_unlock(&wait_thread_mutex);
412
413   pthread_cond_wait(&wait_process,&wait_process_mutex);
414   pthread_mutex_unlock(&wait_process_mutex);
415 }
416
417 extern int __ompi_main(int argc,char **argv);
418
419 void oprc__ompi_main(void)
420 {
421    __ompi_main(ort_argc,ort_argv);
422 }
423
424 int main(int argc, char **argv)
425 {
426    ort_argc=argc;
427    ort_argv=argv;
428    pthread_attr_t tattr;
429    pthread_mutexattr_t mattr;
430    pthread_t tid;
431    int ret, memid, pid;
432    void *arg;
433
434    pthread_mutexattr_init(&mattr);
435    pthread_mutexattr_setpshared(&mattr,PTHREAD_PROCESS_SHARED);
436    pthread_mutex_init(&wait_thread_mutex,&mattr);
437    pthread_mutex_init(&wait_process_mutex,&mattr);
438
439    pthread_cond_init(&wait_thread, NULL);
440    pthread_cond_init(&wait_process, NULL);
441
442    oprc_shmalloc(&seg_addr,(size_t)1024*1024,(int)&memid);
443
444    ret=pthread_attr_init(&tattr);
445    if(ret)
446    {
447       printf("pthread_attr_init error\n");
448       exit(0);
449    }
450
451    ret=pthread_attr_setstack(&tattr,seg_addr,1024*1024);
452    if(ret)
453    {
454       printf("pthread_attr_setstackaddr error\n");
455       exit(0);
456    }
457
458    pthread_mutex_lock(&wait_thread_mutex);
459
460    ret = pthread_create(&tid,&tattr,(void *)oprc__ompi_main,NULL);
```

```
461     if(ret!=0)
462     {
463        printf("pthread_create error %d\n", ret);
464        exit(0);
465     }
466
467     while(!end)
468     {
469        pthread_cond_wait(&wait_thread,&wait_thread_mutex);
470        pthread_mutex_unlock(&wait_thread_mutex);
471        FENCE;
472
473        if(!end)
474        {
475           pid=pfork(number_of_process);
476
477           if(pid!=0)
478           {
479             /* We have to do the following, before the actual execution */
480             ort_get_thread_work(pid, team_argument, NULL, &arg);
481
482             (*team_workfunc)(arg); /* Execute requested code */
483
484             exit(0);
485           }
486        }
487        pthread_mutex_lock(&wait_process_mutex);
488        pthread_mutex_lock(&wait_thread_mutex);
489        pthread_cond_signal(&wait_process);
490        pthread_mutex_unlock(&wait_process_mutex);
491     }
492
493     oprc_shmfree(&memid);
494     oprc_shmfree(&ort_lock_id);
495     oprc_shmfree(&ort_locks_id);
496
497     pthread_attr_destroy(&tattr);
498     pthread_mutex_destroy(&wait_thread_mutex);
499     pthread_mutex_destroy(&wait_process_mutex);
500
501     pthread_cond_destroy(&wait_thread);
502     pthread_cond_destroy(&wait_process);
503
504     exit(0);
505  }
```