ΑΠΟΣΠΑΣΜΑ ΑΠΟ ΤΟ ΒΙΒΛΙΟ

Γρ. Πάντζιου, Β. Μάμαλης, Α. Τομαράς "Εισαγωγή στον Παράλληλο Υπολογισμό: Πρότυπα, Αλγόριθμοι, Προγραμματισμός", Εκδόσεις Νέων Τεχνολογιών, 2013.

Κεφ. 6. Προγραμματισμός Παράλληλων Μηχανών

ΠΕΡΙΕΧΟΜΕΝΑ

6 Προγραμματισμός Παράλληλων Μηχανών

6.1 Πρότυπα Παράλληλου Προγραμματισμού

Η διαδικασία ανάπτυξης παράλληλων προγραμμάτων συνηθίζεται [IPP01] να περιγράφεται στη βάση συγκεκριμένων προτύπων τα οποία στοιχειοθετούνται (τουλάχιστον σαν αρχικό πλαίσιο αναφοράς) ανεξάρτητα συγκεκριμένου υλικού και αρχιτεκτονικής, και περιγράφουν/προσδιορίζουν τις δυνατές επιλογές που έχει ένας προγραμματιστής κατά το σχεδιασμό και ανάπτυξη του παράλληλου προγράμματός του όσον αφορά την τελική (σε φυσικό επίπεδο / επίπεδο κώδικα) αποδόμηση της εφαρμογής του σε διακριτές εργασίες, και τον τρόπο επίδρασης των επιμέρους αυτών διακριτών εργασιών πάνω στο σύνολο των δεδομένων της εφαρμογής. Παρότι τα πρότυπα αυτά είναι θεωρητικά ανεξάρτητα συγκεκριμένου υλικού και αρχιτεκτονικής, η επιλογή του πλέον κατάλληλου προτύπου κάθε φορά από τον προγραμματιστή εξαρτάται σε μεγάλο βαθμό από το διαθέσιμο περιβάλλον/αρχιτεκτονική.

6.1.1 Το Πρότυπο Κοινής Μνήμης

Στο πρότυπο αυτό οι διακριτές εργασίες (*tasks*) στις οποίες αποδομείται ο συνολικός υπολογισμός διαμοιράζονται ένα κοινό χώρο μνήμης, τον οποίο μπορούν να προσπελαύνουν ισότιμα τόσο για ανάγνωση όσο και για εγγραφή/ενημέρωση δεδομένων. Ένα σημαντικό πλεονέκτημα του προτύπου αυτού αποτελεί το γεγονός ότι δεν απαιτείται καμία ειδική πρόνοια για την επικοινωνία (ανταλλαγή δεδομένων) μεταξύ των πολλαπλών διακριτών εργασιών. Κάθε είδους απαιτούμενη επικοινωνία πραγματοποιείται μέσω της κοινής μνήμης, οδηγώντας κατ' αυτόν τον τρόπο σε πιο απλή και ευέλικτη διατύπωση παράλληλων προγραμμάτων.

Λόγω ωστόσο, της κοινής πρόσβασης στον ίδιο σφαιρικό χώρο μνήμης, είναι απαραίτητη η χρήση μηχανισμών συγχρονισμού (κλειδώματα, σημαφόροι, κ.λπ.) τόσο για την προστασία κρίσιμων περιοχών όσο και για την υλοποίηση απαιτήσεων απλής ή υπό συνθήκη αναμονής. Η υποστήριξη επίσης από γλώσσες προγραμματισμού με κατάλληλες επεκτάσεις και αντίστοιχους μεταφραστές είναι απαραίτητη. Το φυσικό περιβάλλον υλοποίησης των διακριτών εργασιών ενός τέτοιου προτύπου είναι ως διεργασίες σε περιβάλλοντα/μηχανές κοινής μνήμης όπου η ταυτοποίηση είναι προφανής. Αντίστοιχες υλοποιήσεις είναι ωστόσο δυνατές και σε περιβάλλοντα κατανεμημένης μνήμης, όπου πραγματοποιείται προσομοίωση της κοινής μνήμης πάνω από τον φυσικά κατανεμημένο στους πολλαπλούς επεξεργαστές χώρο μνήμης του συστήματος, μέσω ειδικού υλικού και λογισμικού.

6.1.2 Το Πολυνηματικό Πρότυπο

Το πολυνηματικό πρότυπο αποτελεί ουσιαστικά μία υποπερίπτωση του προτύπου της κοινής μνήμης. Στο πρότυπο αυτό μία διεργασία (process) μπορεί να συντίθεται από περισσότερες από μία υποδιεργασίες οι οποίες είναι γνωστές ως νήματα/threads (ή αλλιώς ως *ελαφρού* τύπου διεργασίες). Κάθε νήμα αναλαμβάνει να εκτελέσει μία από τις διακριτές εργασίες στις οποίες αποδομείται ο συνολικός υπολογισμός. Κάθε νήμα μπορεί να έχει τα δικά του ιδιωτικά δεδομένα, κληρονομεί ωστόσο και διαμοιράζεται (μπορεί να προσπελαύνει και να χρησιμοποιεί ισότιμα) και το χώρο μνήμης της γονικής διεργασίας καθώς και άλλους πόρους αυτής. Το πολυνηματικό πρότυπο θεωρείται γενικά πιο ευέλικτο, αποδοτικό και οικονομικό σε σχέση με το απλό πρότυπο παράλληλου προγραμματισμού κοινής μνήμης, καθώς σπαταλά κατά κανόνα λιγότερους πόρους, χωρίς να επηρεάζεται η παράλληλη χρονοδρομολογησιμότητα των ανεξάρτητων μεταξύ τους διακριτών εργασιών. Για παράδειγμα μία διεργασία μπορεί να επιτελεί κάποιες επιμέρους λειτουργίες ακολουθιακά και σε κάποιο σημείο να δημιουργεί έναν αριθμό από συνεργαζόμενα νήματα προκειμένου να εκτελεστούν ταυτόχρονα από το σύστημα και να πραγματοποιήσουν παράλληλα έναν σύνθετο υπολογισμό (χωρίς να είναι απαραίτητο το σύστημα να εκχωρήσει σε αυτά το σύνολο των πόρων μίας κανονικής διεργασίας).

Τα νήματα για να επικοινωνήσουν (ανταλλάξουν δεδομένα) μεταξύ τους χρησιμοποιούν τον κοινό χώρο μνήμης της γονικής διεργασίας. Για το λόγο αυτό απαιτείται και εδώ η χρήση μηχανισμών συγχρονισμού (κλειδώματα, σημαφόροι, κ.λπ.) τόσο για την προστασία κρίσιμων περιοχών όσο και για την υλοποίηση απαιτήσεων απλής ή υπό συνθήκη αναμονής. Κλασσικές υλοποιήσεις του πολυνηματικού προτύπου προγραμματισμού αποτελούν η βιβλιοθήκη των Posix Threads (Pthreads library) και το OpenMP API (η χρήση και λειτουργικότητα του οποίου παρουσιάζεται αναλυτικότερα στην επόμενη ενότητα αυτού του κεφαλαίου).

6.1.3 Το Πρότυπο Ανταλλαγής Μηνυμάτων

Στο πρότυπο αυτό οι διακριτές εργασίες στις οποίες αποδομείται ο συνολικός υπολογισμός θεωρούμε ότι έχουν το δικό τους τοπικό χώρο μνήμης καθ' όλη τη διάρκεια της παράλληλης εκτέλεσης. Μπορούν δε να βρίσκονται και να εκτελούνται πολλές τέτοιες (ή/και όλες) διακριτές εργασίες είτε σε ένα μόνο υπολογιστικό σύστημα είτε σε περισσότερα (σαν ξεχωριστές διεργασίες/processes). Η ιδανική κατανομή θα έθετε κάθε διακριτή εργασία σε ξεχωριστό σύστημα/μηχανή, οπότε θα είχαμε και το μέγιστο δυνατό βαθμό παραλληλισμού από άποψη τουλάχιστον φόρτου υπολογισμών.

Για την απαιτούμενη ανταλλαγή δεδομένων μεταξύ τους οι διακριτές εργασίες χρειάζεται να επικοινωνούν με εξωτερικούς μηχανισμούς (καθώς δεν υπάρχει κοινή μνήμη) και ειδικότερα ανταλλάσσοντας μηνύματα (messages). Η ανταλλαγή των δεδομένων σε ένα τέτοιο πρότυπο απαιτεί συνήθως, συμφωνημένης μορφής επικοινωνία. Για να ολοκληρωθεί δηλαδή άρτια και αξιόπιστα μια ανταλλαγή θα πρέπει να εκτελεστεί μία διαδικασία αποστολής από τη μία διακριτή εργασία και μία διαδικασία παραλαβής από την άλλη (είτε σύγχρονα είτε ασύγχρονα).

Κλασσικές υλοποιήσεις του προτύπου αυτού αποτελούν οι βιβλιοθήκες μεταβίβασης μηνυμάτων MPI και PVM με την πρώτη να αποτελεί την πλέον διαδεδομένη και ευρέως χρησιμοποιούμενη (τόσο σε ακαδημαικό όσο και σε εταιρικό επίπεδο) κατά τα τελευταία χρόνια. Η χρήση του MPI για παράλληλο προγραμματισμό κατανεμημένης μνήμης παρουσιάζεται αναλυτικότερα στην τελευταία ενότητα αυτού του κεφαλαίου. Αξίζει να σημειωθεί επίσης, ότι το πρότυπο ανταλλαγής μηνυμάτων μπορεί να συνδυαστεί με το πολυνηματικό πρότυπο (π.χ. συνδυαστική χρήση MPI και OpenMP σε περιβάλλοντα συστοιχιών/clusters ή/και ευρύτερων δικτύων από πολυπύρηνους υπολογιστές), στοιχειοθετώντας κατ' αυτόν τον τρόπο ένα *υβριδικό* πρότυπο παράλληλου προγραμματισμού ιδιαίτερα δημοφιλές στις μέρες μας.

6.1.4 Το Πρότυπο Παραλληλισμού Δεδομένων

Στο πρότυπο αυτό ο συνολικός υπολογισμός αποδομείται κατάλληλα (εφ' όσον το επιτρέπουν οι απαιτήσεις της ίδιας της εφαρμογής) σε διακριτές εργασίες οι οποίες εκτελούν παράλληλα λειτουργίες πάνω σε ένα κοινά προσπελάσιμο σύνολο δεδομένων. Πιο συγκεκριμένα, το σύνολο των δεδομένων είναι συνήθως οργανωμένο και προσφερόμενο στις διακριτές εργασίες σαν μία ενιαία δομή, και κάθε διακριτή εργασία καλείται να επιτελέσει την ίδια λειτουργία αλλά σε διαφορετικό μέρος της δομής αυτής (π.χ. υποθέστε ότι η λειτουργία αυτή μπορεί να είναι για κάθε διακριτή εργασία της μορφής *«πολλαπλασίασε με το πέντε κάθε στοιχείο της δομής»* όπου η δομή να είναι ένας πίνακας ακεραίων). Ο συνολικός υπολογισμός συντίθεται δε συνήθως από πολλαπλές (επαναλαμβανόμενες η μη) τέτοιου είδους λειτουργίες. Υλοποιήσεις του προτύπου είναι δυνατές τόσο σε περιβάλλοντα κοινής όσο και κατανεμημένης μνήμης (είτε αυτόνομα είτε σε συνδυασμό με τα αντίστοιχα πρότυπα στα οποία έχουμε ήδη αναφερθεί), απαιτώντας και εδώ υποστήριξη από γλώσσες προγραμματισμού με κατάλληλες επεκτάσεις και αντίστοιχους μεταφραστές [IPP01].

6.1.5 Πρότυπα Προγραμματισμού σε Υψηλό Επίπεδο

Πέραν των ανωτέρω βασικών προτύπων χαμηλού επιπέδου, η διαδικασία του παράλληλου προγραμματισμού διέπεται και από μία πρόσθετη βασική επιλογή σε υψηλότερο επίπεδο, η οποία αφορά στην απόφαση αν όλες οι διακριτές εργασίες θα κληθούν να εκτελέσουν το ίδιο πρόγραμμα ή όχι. Τα αντίστοιχα πρότυπα υψηλού επιπέδου προσδιορίζουν τον αριθμό και τον τρόπο έκφρασης των διαφορετικών προγραμμάτων που θα αναπτυχθούν, μπορούν δε να εφαρμοστούν κατά κανόνα πάνω από οποιοδήποτε από τα προαναφερόμενα πρότυπα χαμηλού επιπέδου.

6.1.5.1 Το Πρότυπο SPMD (Single Program Multiple Data)

Στο πρότυπο SPMD όλες οι διακριτές εργασίες εκτελούν ένα (το ίδιο) πρόγραμμα ανεξάρτητα από τις λειτουργίες που επιτελούν, αλλά σε διαφορετικό εν δυνάμει σύνολο δεδομένων. Για την υποστήριξη αυτού του προτύπου απαιτείται όλες οι εργασίες να αριθμούνται με μια ξεχωριστή ταυτότητα η κάθε μία (που να την προσδιορίζει μοναδικά καθ' όλη τη διάρκεια εκτέλεσης) και επιπλέον, κάθε εργασία σε χρόνο εκτέλεσης να γνωρίζει ποια είναι η ταυτότητά της. Έτσι, όταν υπαγορεύεται από τις απαιτήσεις της εφαρμογής κάποια διακριτή εργασία να εκτελέσει διαφορετικές λειτουργίες από τις υπόλοιπες (μέσα στο ίδιο πρόγραμμα που εκτελούν όλες), αυτό μπορεί να επιτευχθεί πολύ απλά μέσω μίας κατάλληλης δομής ελέγχου (*if*, *case* κλπ) η οποία να διαφοροποιεί το μονοπάτι εκτέλεσης ανάλογα με την ταυτότητα της εργασίας. Είναι δε καταλληλότερο για υλοποίηση προγραμμάτων που έχουν προκύψει από αποδόμηση η οποία έχει γίνει με βάση τα δεδομένα (*domain decomposition*).

6.1.5.2 Το Πρότυπο MPMD (Multiple Program Multiple Data)

Αντίθετα, στο πρότυπο MPMD οι διακριτές εργασίες μπορούν να εκτελούν διαφορετικά προγράμματα, γραμμένα απευθείας με βάση τις απαιτήσεις της κάθε μίας, και έτσι να μην επιβαρύνονται σε χρόνο εκτέλεσης με την επιλογή μονοπατιού μέσα από μία ή περισσότερες δομές ελέγχου του προγράμματος (πράγμα που εμφανώς καταλήγει σε – έστω και με μικρές διαφορές – αποδοτικότερη συνήθως τελική εκτέλεση). Είναι δε καταλληλότερο για υλοποίηση προγραμμάτων που έχουν προκύψει από αποδόμηση η οποία έχει γίνει με βάση τις λειτουργίες (*functional decomposition*) και όχι με βάση τα δεδομένα (*domain decomposition*). Η ανάπτυξη, διανομή και συντήρηση ωστόσο ενός μόνο προγράμματος είναι αρκετά πιο εύκολα και αποδοτικά διαχειρίσιμη από το λειτουργικό σύστημα ενός παράλληλου συστήματος (ειδικότερα σε περιβάλλοντα κατανεμημένης μνήμης), γεγονός το οποίο έχει οδηγήσει τους περισσότερους κατασκευαστές και προγραμματιστές να επιλέγουν κατά κανόνα την υποστήριξη και χρήση του προτύπου SPMD για τα περιβάλλοντά τους.

6.2 Προγραμματισμός Κοινής Μνήμης με Χρήση OpenMP

6.2.1 Εισαγωγή

Το OpenMP αποτελεί την πρώτη επιτυχημένη προσπάθεια δημιουργίας ενός κοινά αποδεκτού εργαλείου/διεπαφής προγραμματισμού παράλληλων μηχανών κοινής μνήμης (shared memory) σε υψηλό επίπεδο. Μέσω της διεπαφής αυτής καθορίζεται πιο συγκεκριμένα ένα σύνολο από διαταγές (directives) για το μεταφραστή (compiler), οι οποίες προστίθενται στον πηγαίο κώδικα ενός προγράμματος που έχει αρχικά γραφτεί προσανατολισμένο στην ακολουθιακή εκτέλεση. Οι διαταγές αυτές ουσιαστικά καθοδηγούν το μεταφραστή στο πώς να παραλληλοποιήσει τον κώδικα για εκτέλεση σε περιβάλλον πολλαπλών επεξεργαστών κοινής μνήμης.

Ιστορικά, οι πρώτες προσπάθειες για την ανάπτυξη του OpenMP ευωδόθηκαν το 1997, οπότε ανακοινώθηκε και η πρώτη έκδοσή του (έκδοση 1.0), ενώ έχει φτάσει μέχρι σήμερα στις εκδόσεις 3.1/2011 (αποτελεί την τελευταία ενσωματωμένη έκδοση στους αντίστοιχους μεταφραστές) και 4.0/2013 (αποτελεί την τελευταία ανακοινωμένη έκδοση και αναμένεται να ενσωματωθεί στους μεταφραστές μέσα στο επόμενο έτος), οι οποίες περιέχουν πλέον ένα αρκετά ευρύ φάσμα μηχανισμών για τις ανάγκες παραλληλισμού και συγχρονισμού σε περιβάλλον κοινής μνήμης, ακλουθώντας σε χαμηλό επίπεδο το *πολυνηματικό* πρότυπο. Έχει αναπτυχθεί δε και υποστηρίζεται μέχρι σήμερα για τις γλώσσες προγραμματισμού Fortran και C/C++.

Αντίστοιχες προσπάθειες δημιουργίας ευρείας αποδοχής εργαλείων/διεπαφών προγραμματισμού παράλληλων μηχανών κοινής μνήμης σε υψηλό επίπεδο, είχαν γίνει από τους κατασκευαστές πολυεπεξεργαστικών συστημάτων και παλαιότερα (από τις αρχές της δεκαετίας του 1990), χωρίς ωστόσο ιδιαίτερη επιτυχία. Έτσι, για πολλά χρόνια, οι προγραμματιστές σε περιβάλλοντα κοινής μνήμης χρησιμοποιούσαν (και εξακολουθούν να χρησιμοποιούν ακόμα και σήμερα σε σημαντικό βαθμό) για τις ανάγκες τους είτε (α) εξειδικευμένα APIs που παρέχουν μεν δομές υψηλού επιπέδου αλλά είναι μη επαρκώς μεταφέρσιμα, είτε (β) πιο κοινά αποδεκτά APIs (όπως π.χ. τα Posix Threads) τα οποία έχουν υψηλό βαθμό μεταφερσιμότητας αλλά απαιτούν από τον προγραμματιστή να αναλώσει μεγάλη προσπάθεια, όχι μόνο στις ανάγκες παραλληλοποίησης της εφαρμογής του σε υψηλό επίπεδο, αλλά και στις λεπτομέρειες υλοποίησης των μηχανισμών πολυεπεξεργασίας του περιβάλλοντός του σε χαμηλό επίπεδο (δημιουργία και εκκίνηση νημάτων, μοντέλο χρονοδρομολόγησης, συγχρονισμός και επικοινωνία νημάτων, ακριβής σχεδιασμός διαμοίρασης φόρτου και δεδομένων κ.λπ.). Ως απάντηση στο παραπάνω κενό, έρχεται να διεισδύσει σταδιακά

και να καθιερωθεί σε σημαντικό βαθμό κατά τα τελευταία χρόνια το OpenMP, παρέχοντας τα ακόλουθα βασικά πλεονεκτήματα:

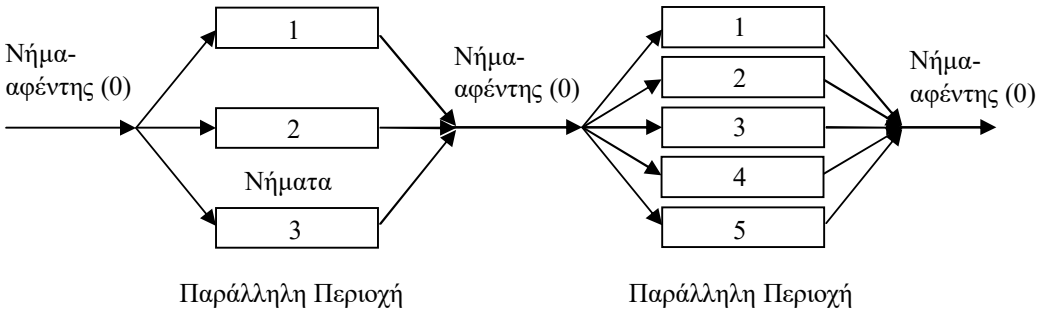
- Βοηθάει τον προγραμματιστή να διατηρήσει το στυλ προγραμματισμού που ακολουθεί σε ακολουθιακό περιβάλλον (και δεν τον εξαναγκάζει να σχεδιάσει εκτενείς ανακατατάξεις στον κώδικά του αυστηρά και μόνο για την παραλληλοποίησή του και ανάλογα με τα εργαλεία που διαθέτει), καθώς του παρέχει επαρκώς αυτόνομες (ξεχωριστές) διαταγές οι οποίες μπορούν να χρησιμοποιηθούν για την παραλληλοποίηση ενός σημαντικού φάσματος αποσπασμάτων κώδικα, προστιθέμενες απλά στον ακολουθιακό κώδικα στα αντίστοιχα σημεία (χωρίς να διαφοροποιείται δηλαδή σημαντικά η μορφή/δομή του προγράμματος στην ακολουθιακή περίπτωση).
- Βοηθάει τον προγραμματιστή να εστιάσει στις ανάγκες παραλληλοποίησης της εφαρμογής του σε υψηλό επίπεδο και να μην είναι αναγκασμένος να ασχοληθεί με λεπτομέρειες υλοποίησης της πολυεπεξεργασίας σε χαμηλό επίπεδο (όπως π.χ. στα Posix Threads).
- Η διαδικασία παραλληλοποίησης από την πλευρά του μεταφραστή είναι πλήρως καθοδηγούμενη από το χρήστη. Ο μεταφραστής δεν χρειάζεται να πραγματοποιήσει κανενός είδους διεξοδική ανάλυση του κώδικα, αλλά αρκεί να στηριχθεί στις πληροφορίες (οδηγίες) που του παρέχει ο χρήστης.
- Γίνεται ολοένα και πιο κοινά αποδεκτό από τους κατασκευαστές πολύ-επεξεργαστικών συστημάτων κοινής μνήμης, εξασφαλίζοντας έτσι μεγάλο βαθμό μεταφερσιμότητας για τα προγράμματα που γράφονται σε αυτό.

6.2.2 Το Πρότυπο Προγραμματισμού του OpenMP

Το OpenMP ακολουθεί σε χαμηλό επίπεδο το *πολυνηματικό* (multithreaded) πρότυπο παράλληλου προγραμματισμού, με τη λογική του μοντέλου παράλληλης εκτέλεσης *fork-join*. Πιο συγκεκριμένα, ένα πρόγραμμα γραμμένο σε OpenMP για περιβάλλον C/C++ αποτελείται στην αρχή (και έτσι ξεκινά την εκτέλεσή του) από ένα νήμα, στο οποίο θα αναφερόμαστε στο εξής ως νήμα-αφέντη (master thread). Το νήμα-αφέντης ξεκινά να εκτελείται ακολουθιακά μέχρι να συναντήσει την πρώτη δήλωση παράλληλης περιοχής (parallel construct). Οι παράλληλες περιοχές δηλώνονται ειδικότερα με το λεκτικό *parallel* και θα αναφερθούμε με περισσότερη λεπτομέρεια στην ακριβέστερη λειτουργικότητα αυτών στη συνέχεια.

Όταν συναντηθεί η πρώτη τέτοια δήλωση παράλληλης περιοχής, το νήμα-αφέντης εκκινεί έναν αριθμό νημάτων (fork), κάθε ένα από τα οποία εκτελεί σαν κώδικά του τον κώδικα που ακολουθεί εντός της παράλληλης περιοχής. Μέσα σε μία παράλληλη περιοχή μπορούν να χρησιμοποιηθούν επίσης από τον προγραμματιστή μεταξύ των άλλων (εκτός δηλαδή από απλό κώδικα ο οποίος θα εκτελείται από όλα τα νήματα) και συγκεκριμένες διαταγές διαμοιρασμού εργασίας (στις οποίες θα αναφερθούμε επίσης αναλυτικότερα στη συνέχεια), μέσω των οποίων επιτυγχάνεται η αποδοτική παραλληλοποίηση συγκεκριμένων μορφών ακολουθιακού κώδικα.

Όταν το σύνολο των νημάτων που δημιουργήθηκαν τελειώσουν την εκτέλεση του κώδικα που υποδεικνύεται από την παράλληλη περιοχή, τερματίζεται η λειτουργία τους (join), και αμέσως μετά τον τερματισμό όλων συνεχίζει μόνο το νήμα-αφέντης με τον κώδικα που ακολουθεί αμέσως μετά την παράλληλη περιοχή. Η παραπάνω μορφή εκτέλεσης μπορεί να επαναληφθεί και παραπάνω από μία φορές μέσα σε ένα παράλληλο πρόγραμμα, όπως φαίνεται και στο σχήμα 1.



Σχήμα 1: Το μοντέλο εκτέλεσης *fork-join* του OpenMP

Το νήμα-αφέντης μετά την εκκίνηση των πολλαπλών νημάτων για την εκτέλεση μιας παράλληλης περιοχής, μετέχει και αυτό ως ενεργό νήμα της ομάδας κατά την περαιτέρω εκτέλεση της παράλληλης περιοχής. Εάν δε το συνολικό πλήθος των νημάτων που δημιουργήθηκαν (συμπεριλαμβανομένου και του νήματος-αφέντη) είναι ίσο με *t*, αυτά αριθμούνται συμβατικά από *0* έως *t-1*, ενώ εξ' ορισμού η ταυτότητα/αριθμός 0 ανατίθεται στο νήμα-αφέντη.

Ο αριθμός των νημάτων που εκκινούνται όταν συναντάται μια παράλληλη περιοχή, ορίζεται αρχικά (αυτόματα από το ίδιο το OpenMP) συνήθως να είναι ίσος με τον αριθμό των επεξεργαστών/πυρήνων που έχει το υπολογιστικό μας σύστημα. Τον αριθμό αυτό ο χρήστης/προγραμματιστής μπορεί να τον αλλάξει (και να ορίσει να εκκινούνται κάθε φορά που συναντάται η εκτέλεση μιας παράλληλης περιοχής,

μεγαλύτερος ή μικρότερος αριθμός νημάτων ανεξαρτήτως του αριθμού επεξεργαστών/ πυρήνων του συστήματος) είτε μέσω της συνάρτησης *omp_set_num_threads()* εντός του κώδικα του προγράμματος είτε καθορίζοντας εξωτερικά τη μεταβλητή περιβάλλοντος OMP_NUM_THREADS.

6.2.3 Βασικές Δομές & Διαταγές Παραλληλισμού

Όλες οι διαταγές (directives) OpenMP προς τον μεταφραστή, σε περιβάλλον C/C++, οι οποίες υποδηλώνουν εντολές παράλληλης εκτέλεσης, αρχίζουν με τα λεκτικά *#pragma omp*, και έχουν την ακόλουθη μορφή:

```
#pragma omp directive [clause ...] new-line
```

Στις επόμενες παραγράφους θα εξετάσουμε τις σημαντικότερες από αυτές, δίνοντας έμφαση στις βασικές διαταγές παραλληλισμού, καθώς και στις διαταγές διαμοίρασης εργασιών, συγχρονισμού και ορισμού διακριτών εργασιών (tasks). Για την ακριβή σύνταξη και τη λειτουργικότητα των αντίστοιχων διαταγών σε περιβάλλον Fortran μπορεί κανείς να ανατρέξει στα [OMP01,OMP02].

6.2.3.1 Η βασική διαταγή παραλληλισμού

Η δήλωση της αρχής μιας παράλληλης περιοχής μπορεί να εισαχθεί σε ένα οποιοδήποτε σημείο του ακολουθιακού μας προγράμματος, και σηματοδοτείται με το λεκτικό/διαταγή *parallel* στην ακόλουθη μορφή:

```
#pragma omp parallel [clause ...] new-line
      structured-block
```

Όπως τονίστηκε και στην προηγούμενη παράγραφο, από τη στιγμή που θα συναντηθεί μια τέτοια διαταγή θα δημιουργηθεί ένας αριθμός από νήματα τα οποία θα εκτελέσουν όλα τον κώδικα που ακολουθεί μέσα στη διαταγή. Ένα απλό παράδειγμα βασικής κατανόησης παρατίθεται παρακάτω:

```
#include <omp.h>

int main(int argc, char **argv)
 {
  #pragma omp parallel
   {
    printf("Hello OpenMP!\n");
   }
 }
```

Αν υποθέσουμε ότι βρισκόμαστε σε ένα υπολογιστικό σύστημα με δύο (2) επεξεργαστές/πυρήνες, τότε μέσω του παραπάνω κώδικα θα δημιουργηθούν αρχικά δύο νήματα τα οποία θα εκτελεστούν παράλληλα και θα τυπώσουν και τα δύο το διαγνωστικό μήνυμα *Hello OpenMP!*

```
Hello OpenMP!

Hello OpenMP!
```

Ακολουθεί παρακάτω ένα λίγο πιο σύνθετο παράδειγμα μέσω του οποίου θα αναφερθούμε σε κάποιες επιπλέον πτυχές της βασικής διαταγής *parallel*.

```
#include <omp.h>

main ()  {

int nthreads, tid;

/* Δυνατότητα επανακαθορισμού του αριθμού νημάτων */
// omp_set_num_threads(4)

/* Δημιουργία των νημάτων */
#pragma omp parallel private(tid)
  {
  /* Βρες ποια είναι η ταυτότητά σου */
  tid = omp_get_thread_num();
  printf("Hello World from thread %d\n", tid);

  /* Εκτελείται μόνο από το νήμα #0 */
  if (tid == 0)
    {
    /* Βρες πόσα νήματα έχουν δημιουργηθεί */
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  } /* Τερματισμός όλων των νημάτων */
}
```

Όπως αναφέρθηκε και στην προηγούμενη παράγραφο, αν υποθέσουμε ότι δημιουργούνται *t* τον αριθμό νήματα, θα αριθμηθούν αυτόματα από το σύστημα με ταυτότητες (*IDs*) από *0* έως *t-1*. Μέσω δε της συνάρτησης *omp_get_thread_num()* κάθε νήμα μπορεί να μάθει ποια είναι η ταυτότητά του. Μέσω επίσης της συνάρτησης *omp_get_num_threads()* μπορεί κάθε νήμα να μάθει ποιος είναι ο συνολικός αριθμός νημάτων που έχουν δημιουργηθεί κατά την εκκίνηση της παράλληλης περιοχής. Και οι δύο παραπάνω πληροφορίες είναι πολύ χρήσιμες για την ευχέρεια διαχείρισης του παραλληλισμού από τον ίδιο τον προγραμματιστή.

Έτσι μέσω του παραπάνω προγράμματος, αν υποθέσουμε πάλι ότι βρισκόμαστε σε ένα παράλληλο σύστημα με δύο επεξεργαστές/πυρήνες, τότε θα δημιουργηθούν και εδώ δύο νήματα τα οποία θα εκτελεστούν παράλληλα και θα τυπώσουν τα ακόλουθα μηνύματα:

```
Hello World from thread 0
Hello World from thread 1
Number of threads = 2
```

Εάν αντίθετα προσθέσουμε την εντολή (κλήση συνάρτησης βιβλιοθήκης) *omp_set_num_threads(4)* πριν την εκκίνηση της παράλληλης περιοχής (στο σημείο δηλαδή του παραπάνω κώδικα όπου έχει ήδη τοποθετηθεί εσκεμμένα σε σχόλια) θα εκκινηθούν τέσσερα (4) νήματα (ανεξάρτητα από τον αριθμό επεξεργαστών/πυρήνων του συστήματος) και θα τυπωθούν τα ακόλουθα:

```
Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
Hello World from thread 3
Number of threads = 4
```

Για να γίνουν τα παραπάνω, παρατηρούμε επίσης ότι δίπλα στη διαταγή *parallel* δηλώνεται ως ιδιωτική (*private)* η μεταβλητή *tid*. Σε αυτήν την μεταβλητή αποθηκεύει κάθε νήμα την ταυτότητά του, και μέσω αυτής αναγνωρίζει παρακάτω κάθε νήμα αν πρέπει ή όχι να μπει μέσα στο μπλοκ της εντολής *if* που ακολουθεί, έτσι ώστε η σχετική πληροφορία (του συνολικού αριθμού νημάτων) να τυπωθεί μόνο από ένα νήμα. Αυτή η μεταβλητή (*tid: private*) λαμβάνει για κάθε νήμα μία συγκεκριμένη τιμή, η οποία δεν συγχέεται με τις αντίστοιχες τιμές που έχει η ίδια μεταβλητή (η μεταβλητή με το ίδιο όνομα) στα υπόλοιπα νήματα.

Αντίστοιχα μπορούμε να δηλώσουμε μία η παραπάνω μεταβλητές να είναι κοινές/διαμοιραζόμενες (*shared*) σε όλα τα νήματα (έτσι ώστε να είναι ισότιμα προσπελάσιμες από όλα τα νήματα και τις αλλαγές που γίνονται από ένα νήμα να μπορούν να τις βλέπουν και τα υπόλοιπα), κάτι το οποίο αποτελεί μία από τις πλέον απαραίτητες δυνατότητες για την αποδοτική διατύπωση και εκτέλεση παράλληλων προγραμμάτων σε περιβάλλοντα κοινής μνήμης.

Πέραν των παραπάνω (*private*, *shared*) σε μια διαταγή *parallel* έχουμε τη δυνατότητα να ορίσουμε/δηλώσουμε (*clause*) επιπλέον (είτε μία είτε περισσότερες από αυτές στη σειρά) τις ακόλουθες παραμέτρους:

```
if (scalar-expression)
firstptivate (list)
num_threads (num)
default (shared | none)
copyin (list)
reduction (operator: list)
```

Μέσω της δήλωσης/παραμέτρου *if* ο προγραμματιστής μπορεί να ορίσει μία συνθήκη, η ικανοποίηση της οποίας (σε χρόνο εκτέλεσης) να είναι απαραίτητη για την εκτέλεση ή όχι της παράλληλης περιοχής (π.χ. αν το πλήθος των δεδομένων εισόδου είναι μεγαλύτερο από ένα όριο να προχωρά το πρόγραμμα σε εκκίνηση νημάτων και παράλληλη εκτέλεση, ειδάλλως να διατηρείται το πρόγραμμα σε ακολουθιακή εκτέλεση). Μέσω της δήλωσης/παραμέτρου *num_threads* μπορεί να καθοριστεί ο αριθμός των νημάτων που επιθυμούμε να εκκινηθεί (ισοδύναμα όπως και με τη συνάρτηση *omp_set_num_threads()*). Μέσω της δήλωσης/παραμέτρου *default* καθορίζεται στο μεταφραστή πώς να αντιμετωπιστούν (ως *shared* ή ως λάθη μετάφρασης) τυχόν μεταβλητές που θα συναντηθούν μέσα στην παράλληλη περιοχή και δεν θα έχουν δηλωθεί ρητά είτε ως *shared* είτε ως *private*. Οι δηλώσεις/παράμετροι *firstprivate* και *copyin* αφορούν σε ειδικότερες δυνατότητες καθορισμού των αρχικών τιμών ιδιωτικών μεταβλητών κατά την εκκίνηση της εκτέλεσης μίας παράλληλης περιοχής, και για περισσότερες λεπτομέρειες όσον αφορά τη χρήση τους μπορεί κανείς να ανατρέξει στα [OMP01,OMP02]. Σχετικά τέλος με τη δήλωση/παράμετρο *reduction* θα αναφερθούμε αναλυτικότερα στην παράγραφο 6.2.6.

6.2.3.1 Οι βασικές διαταγές διαμοιρασμού εργασίας

Οι διαταγές *διαμοιρασμού εργασίας* αποτελούν τον βασικό μηχανισμό του OpenMP για την αυτοματοποιημένη διατύπωση της παραλληλοποίησης ενός καλά καθορισμένου συνόλου προς εκτέλεση λειτουργιών. Μπορούν να τοποθετηθούν μέσα στο μπλοκ/περιοχή μίας διαταγής *parallel* καθορίζοντας πώς ακριβώς θα πρέπει να γίνει η παραλληλοποίηση σε εκείνο το σημείο (για το απόσπασμα δηλαδή του κώδικα στο οποίο τοποθετούνται). Ακολουθεί μία σύντομη περιγραφή για κάθε μία από αυτές (*for*, *sections*, *single*), με αντίστοιχα παραδείγματα εκτέλεσης.

Α. Η διαταγή For

Η βασική διαταγή διαμοιρασμού εργασίας του OpenMP σε περιβάλλον C/C++ είναι η διαταγή *'for'*, μέσω της οποίας μπορεί να διατυπωθεί αυτόματα η παραλληλοποίηση ενός συνόλου επαναλήψεων του προγράμματός μας οι οποίες καθορίζονται μέσα σε μία επαναληπτική εντολή *for* της C/C++. Η διαταγή *for* συντάσσεται στην ακόλουθη βασική μορφή, όπου στο *clause* μπορούν να δηλωθούν διάφορες τιμές παραμέτρων που θα εξετάσουμε παρακάτω:

```
#pragma omp for [clause ...] new-line
      for-loop
```

Έστω για παράδειγμα ότι μας ζητείται να υπολογίσουμε παράλληλα το άθροισμα δύο διανυσμάτων *a* και *b* μήκους *N*, και να αποθηκεύσουμε το αποτέλεσμα στον πίνακα *c*. Η αντίστοιχη επαναληπτική δομή την οποία θα χρησιμοποιούσαμε για την ακολουθιακή υλοποίηση σε C, θα ήταν της μορφής:

```
for (i=0; i < N; i++)
    {
     c[i] = a[i] + b[i];
    }
```

Για να παραλληλοποιήσουμε το ανωτέρω for-loop μέσω της διαταγής διαμοιρασμού εργασιών *for* του OpenMP, αρκεί ένας ελάχιστα διαφοροποιημένος κώδικας της μορφής που ακολουθεί (μέσα στο κώδικα συμπεριλαμβάνονται και οι απαραίτητες εντολές αρχικοποίησης των δύο πινάκων για λόγους αρτιότερης εκτέλεσης και παρουσίασης των αποτελεσμάτων):

```
#include <omp.h>
#define CHUNKSIZE 4
#define N     16

main ()
{

int t, i, tid, chunk;
int a[N], b[N], c[N];

/* Αρχικοποίηση των δύο διανυσμάτων */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 2;
```

```
/* Καθόρισε σε ομάδες των πόσων επαναλήψεων, θα γίνει η
διαμοίραση του for-loop στα τρέχοντα νήματα */
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i,tid)
  {
  tid = omp_get_thread_num();
  #pragma omp for schedule(static,chunk)
  for (i=0; i < N; i++)
    {
     printf("Thread %d computes iteration %d\n", tid, i);
     c[i] = a[i] + b[i];
    }

  }  /* Τερματισμός όλων των νημάτων */

printf("Final Result Adding Vectors:\n");
for (t=0; t < N; t++)
 printf("%d ", c[t]);

}
```

Στο παραπάνω πρόγραμμα, παρατηρούμε καταρχήν ότι πρέπει να δηλωθούν (παράμετροι στη βασική διαταγή *parallel*) απαραίτητα ως κοινές (*shared*) όλες οι μεταβλητές δήλωσης των εμπλεκόμενων διανυσμάτων (*a*, *b* και *c*), καθώς πρέπει αυτά να είναι γνωστά και κοινά προσπελάσιμα (τα *a*, *b* για ανάγνωση και το *c* για εγγραφή) από όλα τα νήματα. Ως κοινή μεταβλητή ορίζεται και η μεταβλητή *chunk* (της οποίας τη χρησιμότητα θα εξετάσουμε στη συνέχεια), ενώ ως ιδιωτικές (*private*) είναι απαραίτητο να δηλωθούν σε κάθε νήμα οι μεταβλητές *i* και *tid*.

Η βασική παράμετρος δε, μέσω της οποίας προσδιορίζεται ο τρόπος παραλληλοποίησης του ακολουθιακού *for-loop* είναι η παράμετρος *schedule*. Μέσω της δήλωσης/παραμέτρου αυτής προσδιορίζεται πιο συγκεκριμένα:

- Ανά ομάδες των πόσων επαναλήψεων θα γίνει η διαμοίραση των επαναλήψεων του *for-loop* στα νήματα που έχουν δημιουργηθεί (αυτό γίνεται καθορίζοντας μία συγκεκριμένη τιμή στη δεύτερη υπο-παράμετρο *chunk*). Αν δεν καθοριστεί συγκεκριμένη τιμή, τότε εξορισμού θεωρείται ότι *chunk*=1 και η διαμοίραση των επαναλήψεων γίνεται μία-προς-μία.
- Αν η διαμοίραση θα γίνει στατικά ή δυναμικά (αυτό γίνεται καθορίζοντας μία συγκεκριμένη τιμή στην πρώτη υπο-παράμετρο, και πιο συγκεκριμένα μία από τις τιμές: *static, dynamic, guided, runtime*). Ορίζοντας διαμοίραση τύπου *static*, οι ομάδες (μήκους *chunk* η κάθε μία) των επαναλήψεων ανατίθενται στα υπάρχοντα

νήματα με προκαθορισμένη σειρά (εναλλάξ και κυκλικά). Δηλαδή, π.χ. αν έχουμε δύο συνολικά νήματα, η πρώτη ομάδα ανατίθεται πάντα στο πρώτο νήμα, η δεύτερη ομάδα πάντα στο δεύτερο, η τρίτη ομάδα στο πρώτο, η τέταρτη ομάδα στο δεύτερο κοκ. Αν αντίθετα οριστεί διαμοίραση τύπου *dynamic*, δεν τηρείται προκαθορισμένη σειρά αλλά αντίθετα οι ομάδες επαναλήψεων ανατίθενται για εκτέλεση στα πολλαπλά νήματα δυναμικά και ανάλογα με τη διαθεσιμότητά τους. Πιο συγκεκριμένα, το πρώτο νήμα που είναι/μένει κάθε φορά ελεύθερο λαμβάνει την επόμενη στη σειρά κάθε φορά ομάδα επαναλήψεων. Ο δεύτερος αυτός τρόπος διαμοίρασης είναι ιδιαίτερα χρήσιμος όταν οι επαναλήψεις που διαμοιράζουμε δεν είμαστε σίγουροι ότι απαιτούν όλες τον ίδιο χρόνο εκτέλεσης. Οι επιλογές *guided* και *runtime* είναι δύο άλλες μορφές δυναμικής διαμοίρασης, για τις οποίες περισσότερες λεπτομέρειες μπορεί να βρει κανείς στα [OMP01,OMP02].

Για παράδειγμα, αν έχουν δημιουργηθεί δύο νήματα, και έχουν καθοριστεί *N=16, chunk=4* (διαμοίραση δηλαδή σε ομάδες των τεσσάρων επαναλήψεων) και διαμοίραση τύπου *static*, όπως φαίνεται στον παραπάνω κώδικα, η διαμοίραση θα γίνει όπως δείχνει η έξοδος του προγράμματος παρακάτω:

```
Thread 0 computes iteration 0
Thread 0 computes iteration 1
Thread 0 computes iteration 2
Thread 0 computes iteration 3
Thread 0 computes iteration 8
Thread 0 computes iteration 9
Thread 0 computes iteration 10
Thread 0 computes iteration 11
Thread 1 computes iteration 4
Thread 1 computes iteration 5
Thread 1 computes iteration 6
Thread 1 computes iteration 7
Thread 1 computes iteration 12
Thread 1 computes iteration 13
Thread 1 computes iteration 14
Thread 1 computes iteration 15

Final Result Adding Vectors:
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
```

Αν αντίθετα οριστεί τιμή *chunk=2* (με διαμοίραση πάλι τύπου *static*) ή έξοδος του προγράμματος θα είναι όπως φαίνεται παρακάτω:

```
Thread 0 computes iteration 0
Thread 0 computes iteration 1
Thread 0 computes iteration 4
Thread 0 computes iteration 5
Thread 0 computes iteration 8
Thread 0 computes iteration 9
Thread 0 computes iteration 12
Thread 0 computes iteration 13
Thread 1 computes iteration 2
Thread 1 computes iteration 3
Thread 1 computes iteration 6
Thread 1 computes iteration 7
Thread 1 computes iteration 10
Thread 1 computes iteration 11
Thread 1 computes iteration 14
Thread 1 computes iteration 15

Final Result Adding Vectors:
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
```

Στο τέλος του μπλοκ/περιοχής της διαταγής *for* υπονοείται εξ' ορισμού για όλα τα νήματα συγχρονισμός τύπου φράγματος, δηλαδή κάθε νήμα (ακόμα και αν έχει ολοκληρώσει τους υπολογισμούς του) πρέπει να περιμένει την ολοκλήρωση όλων των υπολοίπων νημάτων πριν συνεχίσει την εκτέλεσή του.

Πέραν της δήλωσης/παραμέτρου *schedule* σε μια διαταγή διαμοιρασμού εργασιών *for* έχουμε τη δυνατότητα να ορίσουμε/δηλώσουμε (*clause*) επιπλέον (είτε μία είτε περισσότερες από αυτές στη σειρά) τις ακόλουθες παραμέτρους:

```
private (list)
shared (list)
firstptivate (list)
lastprivate (list)
reduction (operator: list)
collapse (n)
```

```
ordered
nowait
```

Οι δηλώσεις/παράμετροι *private* και *shared* έχουν την ίδια έννοια όπως αναφέρθηκε στην προηγούμενη παράγραφο για τη διαταγή *parallel*. Οι δηλώσεις/ παράμετροι *firstprivate*/*lastprivate* αφορούν σε ειδικότερες δυνατότητες καθορισμού των αρχικών/τελικών τιμών ιδιωτικών μεταβλητών κατά την εκκίνηση και κατά το πέρας της εκτέλεσης της διαταγής αντίστοιχα, και για περισσότερες λεπτομέρειες όσον αφορά τη χρήση τους μπορεί κανείς να ανατρέξει στα [OMP01,OMP02]. Μέσω της δήλωσης/παραμέτρου *nowait* γίνεται άρση της αναμονής των νημάτων (υπό μορφή συγχρονισμού φράγματος) στο τέλος της διαταγής μέχρι να τελειώσουν την εκτέλεσή τους όλα τα νήματα. Κάθε νήμα δηλαδή εφόσον τελειώσει την εκτέλεσή του μέσα στη διαταγή *for*, μπορεί να συνεχίσει άμεσα την εκτέλεσή του ανεξάρτητα της προόδου των υπολοίπων. Μέσω της δήλωσης/παραμέτρου *ordered* μπορεί να οριστεί από τον προγραμματιστή η επιθυμία ακολουθιακής εκτέλεσης ορισμένων εντολών του βρόγχου (*for-loop*). Εν συνεχεία μπορεί να οριστεί πιο συγκεκριμένα μέσα στη διαταγή *for* ένα *ordered block* οι εντολές του οποίου (παρότι μέσα σε διαταγή διαμοιρασμού) θα εκτελεστούν σε ακολουθιακή μορφή. Σχετικά με τις δηλώσεις/παραμέτρους *reduction* και *collapse* θα αναφερθούμε αναλυτικότερα στην παράγραφο 6.2.6.

Θα πρέπει να σημειωθεί τέλος ότι η διαταγή διαμοιρασμού *for* μπορεί να συνδυαστεί/ενοποιηθεί (σε περιπτώσεις που καθίσταται αυτό εφικτό από τη δόμηση και τις απαιτήσεις του προγράμματος – δηλαδή π.χ. η παράλληλη περιοχή που θέλουμε να εισάγουμε συντίθεται μόνο από μία διαταγή διαμοιρασμού εργασίας *for*) σαν δήλωση με τη διαταγή *parallel*. Πιο συγκεκριμένα το ζεύγος δηλώσεων

```
#pragma omp parallel …
#pragma omp for …
```

είναι ισοδύναμο (υπό τις συνθήκες που αναφέρθηκαν παραπάνω – για περισσότερες λεπτομέρειες βλ. [OMP02]) με την ακόλουθη ενοποιημένη δήλωση:

```
#pragma omp parallel for …
```

Το ίδιο ισχύει και για τις άλλες δύο διαταγές διαμοίρασης εργασιών (*sections*, *single*) η περιγραφή των οποίων ακολουθεί στις επόμενους παραγράφους.

Β. Η διαταγή Sections

Με τη διαταγή *'sections'* μπορούμε να περιγράψουμε ένα σύνολο από διαφορετικές εργασίες (ενότητες), κάθε μία από τις οποίες επιθυμούμε να εκτελεστεί από ένα μόνο – οποιοδήποτε – νήμα της ομάδας (κάθε μία ακολουθιακά και όλες μαζί παράλληλα). Με τη διαταγή αυτή μπορούμε γενικά να διατυπώσουμε παραλληλισμό σε πιο ασσύμετρη και καθοδηγούμενη μορφή (αναθέτοντας κατά κανόνα διαφορετικό κώδικα/εργασία σε κάθε νήμα), αντίθετα με τη διαταγή *for* όπου η διατύπωση/ παραλληλοποίηση ήταν απόλυτα συμμετρική. Η διαταγή *sections* συντάσσεται στην ακόλουθη βασική μορφή, όπου στο *clause* μπορούν να δηλωθούν διάφορες τιμές παραμέτρων τις οποίες θα εξετάσουμε παρακάτω:

```
#pragma omp sections [clause ...] new-line
{
 #pragma omp section new-line
      structured-block
 #pragma omp section new-line
      structured-block
 ...
}
```

Για παράδειγμα αν μας ζητείται να υπολογίσουμε παράλληλα τα ακόλουθα δύο αθροίσματα: (α) το άθροισμα δύο διανυσμάτων *a* και *b* (μήκους *N*), και (β) το άθροισμα δύο άλλων διανυσμάτων *e* και *f* (επίσης μήκους *N*), θα μπορούσαμε εναλλακτικά ως προς τη χρήση της διαταγής *for* που είδαμε παραπάνω για καθένα από αυτά (και θεωρώντας αρχικά ότι δεν έχουμε παραπάνω από δύο επεξεργαστές/πυρήνες στο υπολογιστικό μας σύστημα), να διατυπώσουμε τον παράλληλο κώδικα που ακολουθεί με χρήση της διαταγής *sections* ως εξής:

```
#include <omp.h>
#define N     10

main ()
{

int i,t;
int a[N],b[N],c[N],d[N],e[N],f[N];

/* Αρχικοποιήσεις διανυσμάτων */
for (i=0; i < N; i++) {
  a[i] = i * 2;
  b[i] = i * 3;
  e[i] = i * 4;
```

```
  f[i] = i * 5;

  }

#pragma omp parallel shared(a,b,c,d,e,f) private(i)
  {

  #pragma omp sections
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = e[i] + f[i];

    }  /* τέλος της διαταγής sections */

  }  /* τέλος της παράλληλης περιοχής */

    printf("Final Result Vector C:\n");
    for (t=0; t < N; t++)
      printf("%d ", c[t]);
    printf("Final Result Vector D:\n");
    for (t=0; t < N; t++)
      printf("%d ", d[t]);
}
```

Ανεξάρτητα από το πόσα νήματα έχουν δημιουργηθεί συνολικά, το πρώτο *section* (άθροισμα) θα εκτελεστεί ακολουθιακά από το πρώτο νήμα και το δεύτερο *section* (άθροισμα) επίσης ακολουθιακά από το δεύτερο νήμα. Μεταξύ τους ωστόσο οι δύο εκτελέσεις θα γίνουν παράλληλα. Αν έχουν επίσης τυχόν δημιουργηθεί παραπάνω από δύο νήματα, τα υπόλοιπα νήματα (πέραν των δύο πρώτων) θα παραμείνουν ανενεργά.

Γενικότερα, αν έχουμε στη διάθεσή μας μεγαλύτερο αριθμό νημάτων/ επεξεργαστών από *sections*, πάντα ένας αριθμός νημάτων (τα πλεονάζοντα στον αριθμό) θα μένουν ανενεργά, ενώ αν έχουμε στη διάθεσή μας μεγαλύτερο αριθμό *sections* από νήματα/επεξεργαστές, τα πολλαπλά *sections* θα ανατίθενται στα υπάρχοντα νήματα ανάλογα με τη διαθεσιμότητά των τελευταίων (δηλ. το κάθε επόμενο στη σειρά *section* στο επόμενο στη σειρά διαθέσιμο νήμα). Αν έχουμε π.χ. τρία *sections* και έχουν δημιουργηθεί δύο νήματα, το πρώτο *section* θα ανατεθεί στο πρώτο νήμα, το δεύτερο *section* στο δεύτερο νήμα και το τρίτο *section* σε όποιο από τα δύο νήματα τελειώσει πρώτο την εργασία του.

Σε κάθε περίπτωση (όπως και στη διαταγή διαμοιρασμού εργασίας *for*) υπονοείται εξ' ορισμού και εδώ ένας συγχρονισμός τύπου φράγματος στο τέλος του μπλοκ/περιοχής της διαταγής *sections*, δηλαδή κάθε νήμα (είτε πραγματοποιεί υπολογισμούς είτε όχι) θα πρέπει να περιμένει την ολοκλήρωση όλων των ενεργών νημάτων/εργασιών της διαταγής πριν συνεχίσει την εκτέλεσή του.

Σε μία διαταγή διαμοιρασμού εργασιών *sections* έχουμε τη δυνατότητα επίσης να ορίσουμε/δηλώσουμε (*clause*) τις ακόλουθες παραμέτρους (είτε μία είτε περισσότερες από αυτές στη σειρά):

```
private (list)
firstptivate (list)
lastprivate (list)
reduction (operator: list)
nowait
```

Η δήλωση/παράμετρος *private* έχει την ίδια έννοια όπως αναφέρθηκε στην προηγούμενη παράγραφο για τη διαταγή *parallel*. Οι δηλώσεις/παράμετροι *firstprivate*/*lastprivate* αφορούν σε ειδικότερες δυνατότητες καθορισμού των αρχικών/τελικών τιμών ιδιωτικών μεταβλητών κατά την εκκίνηση και το πέρας της εκτέλεσης της διαταγής αντίστοιχα, και για περισσότερες λεπτομέρειες μπορεί κανείς να ανατρέξει στα [OMP01,OMP02]. Μέσω της δήλωσης/παραμέτρου *nowait* γίνεται άρση της αναμονής των νημάτων (υπό μορφή συγχρονισμού φράγματος) στο τέλος της διαταγής μέχρι να τελειώσουν την εκτέλεσή τους όλα τα νήματα. Κάθε νήμα δηλαδή εφόσον τελειώσει την εκτέλεσή του μέσα στη διαταγή *sections* (ή αν δεν του είχε ανατεθεί ούτως ή άλλως καμία εργασία μέσα σε αυτή τη διαταγή), μπορεί να συνεχίσει άμεσα την εκτέλεσή του ανεξάρτητα της προόδου των υπολοίπων. Σχετικά με τη δήλωση/παράμετρο *reduction* θα αναφερθούμε αναλυτικότερα στην παράγραφο 6.2.6.

Θα πρέπει τέλος να σημειωθεί ότι η διαταγή *sections* μπορεί υπό συνθήκες να χρησιμοποιηθεί και για την υποστήριξη μη δομημένου παραλληλισμού (έστω και με όχι ιδιαίτερα αποδοτικό τρόπο), και ειδικότερα για τη διατύπωση αναδρομικών κλήσεων όπως π.χ. περιγράφεται στο [AC09] και στην παράγραφο 6.2.6.4.

Γ. Η διαταγή Single

Με τη διαταγή *'single'* μπορούμε να καθορίσουμε ένα απόσπασμα κώδικα μέσα στο μπλοκ/περιοχή μίας διαταγής *parallel*, να εκτελεστεί από ένα μόνο (το

πρώτο που θα είναι διαθέσιμο εκείνη τη στιγμή) νήμα της ομάδας νημάτων που έχουν δημιουργηθεί (σαν να εκτελείται δηλαδή ακολουθιακά). Η διαταγή *single* συντάσσεται στην ακόλουθη βασική μορφή, όπου στο *clause* μπορούν να δηλωθούν διάφορες τιμές παραμέτρων τις οποίες θα εξετάσουμε παρακάτω:

```
#pragma omp single [clause ...] new-line
      structured-block
```

Για παράδειγμα, αν εκτελέσουμε το ακόλουθο πρόγραμμα και θεωρήσουμε ότι έχουν δημιουργηθεί με την εκκίνηση της παράλληλης περιοχής (`#pragma omp parallel`) συνολικά τέσσερα νήματα, η συνάρτηση *Do_Job1()* θα εκτελεστεί τέσσερις φορές, η συνάρτηση *Do_Job2()* θα εκτελεστεί μία φορά, και η συνάρτηση *Do_Job3()* θα εκτελεστεί επίσης τέσσερις φορές.

```
#pragma omp parallel
 {
   Do_Job1();
   #pragma omp single
   {
     Do_Job2();
   }
   Do_Job3();
 }
```

Θα πρέπει να σημειωθεί ότι και εδώ (όπως και στις δύο προηγούμενες διαταγές διαμοίρασης εργασιών – *for* και *sections*) υπονοείται εξ' ορισμού ένας συγχρονισμός τύπου φράγματος στο τέλος του μπλοκ/περιοχής της διαταγής *single*, δηλαδή, όλα τα νήματα θα πρέπει να περιμένουν την ολοκλήρωση της διαταγής *single* πριν συνεχίσουν την εκτέλεσή τους. Μπορεί ωστόσο και εδώ να γίνει άρση αυτής της αναμονής αν εισαχθεί στη γραμμή δήλωσης της *single* η παράμετρος *nowait*.

Η επιλογή αυτή είναι ιδιαίτερα χρήσιμη ειδικά σε αυτήν την περίπτωση που ξέρουμε εκ των προτέρων ότι τα υπόλοιπα (εκτός ενός) νήματα δεν κάνουν καμία εργασία (εκτός αν για την ορθή συνέχειά τους απαιτείται το αποτέλεσμα της εργασίας που επιτελείται μέσα στη διαταγή *single*). Για παράδειγμα, φανταστείτε ότι η συνάρτηση *Do_Job1()* μπορεί να αφορά στην παράλληλη *άθροιση* των στοιχείων ενός διανύσματος (και να υλοποιείται με μια αντίστοιχη διαταγή διαμοιρασμού *for*), στη συνέχεια η *Do_Job2()* να αφορά στον υπολογισμό της *μέσης τιμής* των στοιχείων του διανύσματος (διαίρεση του αθροίσματος με το *N*, η οποία αρκεί και πρέπει να γίνει από ένα μόνο νήμα), και τέλος η *Do_Job3()* να αφορά στον παράλληλο υπολογισμό

των *αποκλίσεων* των στοιχείων του διανύσματος από την υπολογισθείσα παραπάνω μέση τιμή (και να υλοποιείται και αυτή με μια αντίστοιχη διαταγή διαμοιρασμού *for*).

Γενικά, σε μία διαταγή διαμοιρασμού εργασιών *single* έχουμε τη δυνατότητα να ορίσουμε/δηλώσουμε (*clause*) τις ακόλουθες παραμέτρους (είτε μία είτε περισσότερες από αυτές στη σειρά):

```
private (list)
firstptivate (list)
copyprivate (list)
nowait
```

Η δήλωση/παράμετρος *private* έχει την ίδια έννοια όπως αναφέρθηκε στην προηγούμενη παράγραφο για τη διαταγή *parallel*. Οι δηλώσεις/παράμετροι *firstprivate* και *copyprivate* αφορούν σε ειδικότερες δυνατότητες καθορισμού των τιμών των ιδιωτικών μεταβλητών τόσο του νήματος που εκτελεί τη διαταγή όσο και των υπολοίπων, και για περισσότερες λεπτομέρειες όσον αφορά τη χρήση τους μπορεί κανείς να ανατρέξει στα [OMP01,OMP02]. Μέσω τέλος της δήλωσης/παραμέτρου *nowait* γίνεται (όπως αναφέρθηκε και παραπάνω) άρση της αναμονής των νημάτων στο τέλος της διαταγής μέχρι να τελειώσουν την εκτέλεσή τους όλα τα νήματα.

Θα πρέπει να σημειωθεί επίσης ότι η διαταγή *single* μπορεί υπό συνθήκες να χρησιμοποιηθεί (συνδυαζόμενη με τη δήλωση *nowait*) και για την υποστήριξη μη δομημένου παραλληλισμού (έστω και με μη ιδιαίτερα αποδοτικό τρόπο) [AC09], όπως για παράδειγμα σε κάποιες από τις περιπτώσεις που παρουσιάζονται στη συνέχεια του κεφαλαίου για τη χρήση της διαταγής task (βλ. παρ. 6.2.5).

6.2.4 Διαταγές Υποστήριξης Συγχρονισμού

Το OpenMP διαθέτει μηχανισμούς (διαταγές) τόσο για την υποστήριξη αμοιβαίου αποκλεισμού (mutual exclusion) όσο και για υλοποίηση συγχρονισμού φράγματος (barrier). Δεν διαθέτει ειδικότερο μηχανισμό για την υλοποίηση απλής ή υπό συνθήκη αναμονής (condition synchronization), μπορεί ωστόσο για τους σκοπούς αυτούς να χρησιμοποιηθούν σε μεγάλο βαθμό οι μηχανισμοί κλειδώματος (locks) που παρέχει σε μορφή συναρτήσεων βιβλιοθήκης χρόνου εκτέλεσης.

6.2.4.1 Η διαταγή *critical*

Η διαταγή critical αποτελεί τη βασική διαταγή αμοιβαίου αποκλεισμού που παρέχει το OpenMP και συντάσσεται ως ακολούθως:

```
#pragma omp critical [(name)] new-line
      structured-block
```

Προκειμένου να δούμε τη χρήση της διαταγής critical με ένα απλό παράδειγμα ας θυμηθούμε πρώτα την έννοια της κρίσιμης περιοχής. Μία από τις απλούστερες (και γνωστή σε όλους μας) κρίσιμη περιοχή αποτελεί η αύξηση (increment) μίας κοινής μεταβλητής. Για παράδειγμα η εκτέλεση της πράξης *x=x+1* (όπου *x* μία κοινή μεταβλητή) αν επιτραπεί να εκτελεστεί παράλληλα/ταυτόχρονα από παραπάνω από ένα νήματα χωρίς προστασία (αμοιβαίο αποκλεισμό), είναι πιθανό να δώσει λανθασμένο τελικό αποτέλεσμα (με το πέρας δηλαδή της εκτέλεσης όλων των νημάτων η τελική τιμή του x είναι πιθανό να μην έχει αυξηθεί τόσες φορές όσα είναι τα νήματα που εκτέλεσαν την εντολή αλλά αρκετά λιγότερες). Αυτό συμβαίνει γιατί η εντολή *x=x+1* είναι μία σύνθετη (μη ατομική) εντολή η οποία στην πραγματικότητα συντίθεται (σε επίπεδο συμβολικής γλώσσας και γλώσσας μηχανής ενός επεξεργαστή) από τρεις διαφορετικές ατομικές εντολές της παρακάτω μορφής:

Νήμα 1	**Νήμα 2**
load *x*	**load** *x*
add *1*	**add** *1*
store *x*	**store** *x*

Έτσι, αν για παράδειγμα επιχειρήσουν παραπάνω από ένα νήματα (έστω δύο όπως φαίνεται στο παραπάνω παράδειγμα) να εκτελέσουν ταυτόχρονα/παράλληλα τις παραπάνω εντολές, υπάρχει περίπτωση να διαβάσουν και τα δύο (π.χ. αν εκτελέσουν την εντολή *load* το ένα αμέσως μετά το άλλο) την ίδια τιμή του *x*, πριν κάποιο από τα δύο προλάβει να εκτελέσει την εντολή *add* και να την αυξήσει, και έτσι τελικά να αποθηκευτεί στο *x* τελική τιμή (μέσω των εντολών *store* που θα εκτελέσουν και τα δύο νήματα στη συνέχεια) αυξημένη μόνο κατά 1 και όχι κατά 2 όπως θα ήταν το σωστό. Το παραπάνω λάθος μπορεί να αναπαραχθεί ανεξέλεγκτα αν τα νήματα εκτελούν τις ανωτέρω εντολές επαναληπτικά.

Ενδεικτικά, δίνεται παρακάτω ένα ελάχιστο παράλληλο πρόγραμμα σε OpenMP στο οποίο εκτελείται η πράξη *x=x+1* επαναληπτικά (1000 φορές σε ομάδες των 10 επαναλήψεων) και χωρίς καμία προστασία (αμοιβαίο αποκλεισμό).

```
#include <omp.h>
#define CHUNKSIZE 10
#define N     1000

main ()
{

int i, chunk;
int x=0;

chunk = CHUNKSIZE;

#pragma omp parallel private(i)
  {
  #pragma omp for schedule(static,chunk)
  for (i=0; i < N; i++)
    {
      x=x+1;
    }

  }  /* Τερματισμός όλων των νημάτων */

printf("%d\n", x);

}
```

Αν τρέξουμε το παραπάνω πρόγραμμα σε ένα σύστημα με παραπάνω από έναν επεξεργαστές/πυρήνες (αρκούν δύο - και αντίστοιχος αριθμός νημάτων που θα έχουν αρχικοποιηθεί μέσω της διαταγής *parallel*), είναι σχεδόν σίγουρο (για τους παραπάνω λόγους) ότι θα μας επιστρέψει λανθασμένο αποτέλεσμα (δηλαδή τελική τιμή του *x* μικρότερη του 1000). Δοκιμάστε το!

Η παραπάνω δυσλειτουργία, η οποία προκαλείται λόγω της ανεξέλεγκτης ανάμιξης (interference) εντολών μεταξύ των πολλαπλών νημάτων που εκτελούν ταυτόχρονα την ίδια κρίσιμη περιοχή, μπορεί να αποφευχθεί με κατάλληλη εφαρμογή της διαταγής *critical* όπως φαίνεται στο πρόγραμμα που ακολουθεί.

```
#include <omp.h>
#define CHUNKSIZE 10
#define N     1000

main ()
{

int i, chunk;
int x=0;
```

```
chunk = CHUNKSIZE;

#pragma omp parallel private(i)
  {
  #pragma omp for schedule(static,chunk)
  for (i=0; i < N; i++)
    {
     #pragma omp critical (inc_x)
       {
        x=x+1;
       }
    }

  }  /* Τερματισμός όλων των νημάτων */

printf("%d\n", x);

}
```

Εάν τρέξουμε πλέον το παραπάνω πρόγραμμα (με προστατευόμενο δηλαδή των κώδικα αύξησης της κοινής μεταβλητής *x* μέσω της δήλωσης *critical*) σε ένα σύστημα με παραπάνω από έναν επεξεργαστές/πυρήνες, είναι σίγουρο ότι θα μας βγάλει στο τέλος το σωστό αποτέλεσμα ($x = 1000$). Το ίδιο αποτελεσματική θα ήταν επίσης η προστασία της κρίσιμης περιοχής αν αυτή αποτελείτο (όπως είναι και το σύνηθες) και από περισσότερες από μία (απλές ή σύνθετες) εντολές.

Σε παρένθεση, δίπλα στη δήλωση *critical*, μπορούμε προαιρετικά να εισάγουμε ένα όνομα το οποίο θα αντιστοιχεί στην κρίσιμη περιοχή την οποία προστατεύουμε. Δηλώσεις *critical* με το ίδιο όνομα αντιμετωπίζονται από το σύστημα ως η ίδια κρίσιμη περιοχή, ενώ αν έχουν διαφορετικό όνομα αντιμετωπίζονται ως διαφορετικές. Όλες οι δηλώσεις *critical* επίσης στις οποίες δεν έχει δοθεί όνομα αντιμετωπίζονται ως μία (η ίδια) κρίσιμη περιοχή. Τέλος, θα πρέπει να σημειωθεί ότι η δήλωση critical έχει τον περιορισμό ότι δεν είναι εφικτή η μεταπήδηση (*jump*) από μέσα προς τα έξω ή ανάποδα (από έξω προς τα μέσα) στον κώδικα της κρίσιμης περιοχής που ορίζεται μέσα στο σώμα της.

6.2.4.2 Η διαταγή *atomic*

Το OpenMP παρέχει επίσης έναν πιο απλό και γρήγορο μηχανισμό προστασίας κρίσιμων περιοχών (αμοιβαίου αποκλεισμού) ο οποίος μπορεί να χρησιμοποιηθεί για τη διασφάλιση της ατομικότητας απλών εντολών ενημέρωσης (αλλαγής) μιας κοινής μεταβλητής εντός μίας παράλληλης περιοχής. Ο μηχανισμός αυτός παρέχεται μέσω της διαταγής *atomic* και συντάσσεται ως ακολούθως:

```
#pragma omp atomic new-line
      expression-stmt
```

Το *expression-stmt* είναι μία έκφραση η οποία μπορεί να έχει μία από τις ακόλουθες μορφές:

x <*op*> = *expr*

x + +

+ + *x*

x - -

- - *x*

όπου *x* μία κοινή μεταβλητή και <*op*> ένας από τους ακόλουθους τελεστές:

+, *, -, /, &, ^, |, <<, >>

Για παράδειγμα αν υποθέσουμε ότι η κρίσιμη περιοχή στον κώδικα/ πρόγραμμα της προηγούμενης παραγράφου (όπου είδαμε τη χρήση της διαταγής *critical*) αποτελείται μόνο από την εντολή *x=x+1*, θα μπορούσαμε να διασφαλίσουμε τον αμοιβαίο αποκλεισμό κατά την ενημέρωσή της, πιο απλά και γρήγορα (μέσω της διαταγής *atomic*) όπως φαίνεται στον κώδικα που ακολουθεί.

```
#include <omp.h>
#define CHUNKSIZE 10
#define N     1000

main ()
{

int i, chunk;
int x=0;

chunk = CHUNKSIZE;

#pragma omp parallel private(i)
  {
  #pragma omp for schedule(static,chunk)
  for (i=0; i < N; i++)
    {
     #pragma omp atomic
        x++;
    }

  }  /* Τερματισμός όλων των νημάτων */
```

```
printf("%d\n", x);
}
```

Εάν τρέξουμε το παραπάνω πρόγραμμα σε ένα σύστημα με παραπάνω από έναν επεξεργαστές/πυρήνες, είναι και εδώ σίγουρο ότι θα μας δώσει στο τέλος το σωστό αποτέλεσμα ($x = 1000$). Εάν ωστόσο η κρίσιμη περιοχή την οποία επιθυμούμε να προστατεύσουμε αποτελείται από περισσότερες από μία εντολές είναι προφανές ότι δεν αρκεί να χρησιμοποιήσουμε την εντολή *atomic* αλλά θα πρέπει αντίθετα να χρησιμοποιήσουμε κάποιον από τους άλλους δύο παρεχόμενους μηχανισμούς συγχρονισμού του OpenMP (είτε τη διαταγή *critical* ή το μηχανισμό *κλειδώματος* τον οποίον θα παρουσιάσουμε στη συνέχεια). Το ίδιο επίσης ισχύει αν πρόκειται για μία μεν αλλά πιο σύνθετη (σε σχέση με τις παραπάνω αναφερόμενες) εντολή/πράξη εκχώρησης με μία ή περισσότερες κοινές μεταβλητές.

6.2.4.3 Μηχανισμοί κλειδώματος

Το OpenMP παρέχει επίσης έναν γενικότερο μηχανισμό κλειδώματος, μέσω αντίστοιχων συναρτήσεων βιβλιοθήκης, ο οποίος μπορεί να χρησιμοποιηθεί για την υλοποίηση κάθε είδους (απλών αλλά και πιο σύνθετων) αναγκών αμοιβαίου αποκλεισμού, χωρίς μάλιστα τους περιορισμούς που εισάγει η χρήση της διαταγής *critical*. Μπορεί επίσης να χρησιμοποιηθεί σε μεγάλο βαθμό και για πιο σύνθετες ανάγκες συγχρονισμού/επικοινωνίας μεταξύ των πολλαπλών νημάτων (όπως θα δούμε αναλυτικότερα στην παράγραφο 6.2.6). Οι βασικές συναρτήσεις κλειδώματος οι οποίες υποστηρίζουν τον ανωτέρω μηχανισμό κλειδώματος είναι οι ακόλουθες:

```
void omp_init_lock(omp lock t *lock);
```

Μέσω της συνάρτησης αυτής αρχικοποιείται ένα κλείδωμα (και τίθεται αρχικά να είναι ελεύθερο/ξεκλείδωτο προς δέσμευση).

```
void omp_set_lock(omp lock t *lock);
```

Μέσω της συνάρτησης αυτής επιχειρείται η δέσμευση ενός κλειδώματος. Αν είναι ελεύθερο δεσμεύεται και συνεχίζει η εκτέλεση του νήματος κανονικά, αν δεν είναι ελεύθερο (είναι δηλαδή ήδη δεσμευμένο από ένα άλλο νήμα) τότε το νήμα που κάλεσε τη συνάρτηση μπλοκάρεται μέχρι το κλείδωμα να ελευθερωθεί.

```
void omp_unset_lock(omp lock t *lock);
```

Μέσω της συνάρτησης αυτής ελευθερώνεται ένα κλείδωμα από το νήμα που το κατέχει. Εάν κάποιο/α από τα υπόλοιπα νήματα είχε/αν επιχειρήσει νωρίτερα να δεσμεύσει/ουν το κλείδωμα και έχει/ουν μπλοκαριστεί επειδή ήταν δεσμευμένο, επιλέγεται ένα από αυτά προκειμένου να ξαναπροσπαθήσει.

Με βάση τις ανωτέρω συναρτήσεις βιβλιοθήκης μπορούμε να προστατέψουμε οποιαδήποτε κρίσιμη περιοχή (η οποία αποτελείται από μία ή περισσότερες απλές ή σύνθετες εντολές), θέτοντας απλά κάθε νήμα που επιθυμεί να εισέλθει στην κρίσιμη περιοχή να επιχειρεί (μέσω της συνάρτησης *omp_set_lock()*) να δεσμεύσει το κλείδωμα που αντιστοιχεί σε αυτήν την κρίσιμη περιοχή (και μόνο εφόσον το δεσμεύσει να μπορεί να προχωρήσει), και όταν εξέρχεται από την κρίσιμη περιοχή να ελευθερώνει το κλείδωμα (μέσω της συνάρτησης *omp_unset_lock()*). Έτσι, αν υποθέσουμε ότι η κρίσιμη περιοχή στον κώδικά μας είναι η αύξηση της κοινής μεταβλητής *x* (*x=x+1*) όπως και στις προηγούμενες παραγράφους (όπου είδαμε τη χρήση των διαταγών *critical* και *atomic*), η προστασία αυτής θα μπορούσε να επιτευχθεί ισοδύναμα με το ακόλουθο πρόγραμμα:

```
#include <omp.h>
#define CHUNKSIZE 10
#define N     1000

main ()
{

int i, chunk;

omp_lock_t lock1;

omp_init_lock(&lock1);

chunk = CHUNKSIZE;

#pragma omp parallel private(i)
  {
  #pragma omp for schedule(static,chunk)
  for (i=0; i < N; i++)
    {
     omp_set_lock(&lock1);
        x=x+1;
     omp_unset_lock(&lock1);
       }
    }

  }  /* Τερματισμός όλων των νημάτων */

printf("%d\n", x);
```

```
}
```

Εάν θέλουμε να υλοποιήσουμε περισσότερα από ένα κλειδώματα στο ίδιο πρόγραμμα (για την προστασία π.χ. παραπάνω από μία κρίσιμων περιοχών) δεν έχουμε παρά να αρχικοποιήσουμε αντίστοιχο αριθμό μεταβλητών κλειδώματος (lock1, lock2, ...) και να χρησιμοποιήσουμε κάθε μία από αυτές για την προστασία κάθε διαφορετικής κρίσιμης περιοχής. Δύο επιπλέον χρήσιμες συναρτήσεις σχετικές με το μηχανισμό κλειδώματος του OpenMP είναι οι ακόλουθες:

```
void omp_destroy_lock(omp lock t *lock);
```

Μέσω της συνάρτησης αυτής απελευθερώνονται οι πόροι που καταλαμβάνει ένα αρχικοποιημένο κλείδωμα.

```
void omp_test_lock(omp lock t *lock);
```

Μέσω της συνάρτησης αυτής δίνεται η δυνατότητα σε ένα νήμα να επιχειρήσει να δεσμεύσει ένα κλείδωμα χωρίς ωστόσο να μπλοκάρεται στην περίπτωση που το κλείδωμα είναι ήδη δεσμευμένο.

Το OpenMP παρέχει επίσης έναν ακόμα μηχανισμό κλειδώματος, τα *nestlocks*, τα οποία διαφέρουν σε σχέση με τα απλά κλειδώματα στο γεγονός ότι όταν ένα νήμα επιχειρεί να δεσμεύσει ένα κλείδωμα που είναι δεσμευμένο, μπλοκάρεται μόνο αν δεν είναι το ίδιο νήμα που έχει δεσμεύσει το κλείδωμα. Η χρήση των κλειδωμάτων της μορφής αυτής είναι πέραν του σκοπού του κεφαλαίου αυτού, και για περισσότερες λεπτομέρειες μπορεί να ανατρέξει κανείς στα [OMP01,OMP02].

6.2.4.4 Η διαταγή *barrier*

Το OpenMP παρέχει επίσης στον προγραμματιστή τη δυνατότητα αυτοματοποιημένης υλοποίησης καθολικής αναμονής τύπου *φράγματος* του συνόλου των νημάτων που έχουν αρχικοποιηθεί σε μία παράλληλη περιοχή, σε οποιοδήποτε *ελεύθερο* σημείο αυτής (εκτός δηλ. από το εσωτερικό κάποιας διαταγής διαμοιρασμού εργασίας ή το εσωτερικό των διαταγών *master*, *critical* και *ordered*). Η δυνατότητα αυτή παρέχεται μέσω της διαταγής *barrier* η οποία συντάσσεται ως ακολούθως:

```
#pragma omp barrier new-line
```

Μία συχνή χρήση της διαταγής *barrier* σε προγράμματα OpenMP είναι πριν την είσοδο σε μία διαταγή διαμοίρασης εργασίας (π.χ. for, sections). Πριν την είσοδο

σε μια διαταγή διαμοίρασης εργασίας δεν υλοποιείται εσωτερικά από το ίδιο το OpenMP αναμονή τύπου φράγματος, δηλαδή κάθε νήμα μπορεί να εισέλθει και να αρχίσει την εκτέλεση του δικού τους μέρους εργασιών σε διαφορετική χρονική στιγμή (π.χ. πριν τα υπόλοιπα ή κάποια από τα υπόλοιπα νήματα να έχουν ολοκληρώσει τις εργασίες τους έξω από την διαταγή αυτή). Αντίθετα, όπως είδαμε στις προηγούμενες παραγράφους υλοποιείται από το ίδιο το OpenMP τέτοιου είδους συγχρονισμός (barrier) στο τέλος μιας διαταγής διαμοίρασης εργασίας (εκτός αν έχουμε χρησιμοποιήσει τη δήλωση *nowait* για την άρση του).

Έτσι, αν για τη διασφάλιση της ορθότητας υλοποίησης των παράλληλων υπολογισμών μίας διαταγής διαμοίρασης εργασίας, θέλουμε να είμαστε σίγουροι ότι πριν την εκκίνησή της από οποιοδήποτε νήμα έχουν ολοκληρώσει όλα τα νήματα την εργασία τους έξω από αυτήν (έστω π.χ ότι πριν την είσοδο στην διαταγή διαμοίρασης κάποια από τα νήματα ενημερώνουν κάποιες κοινές δομές και μεταβλητές της εφαρμογής τις οποίες θα χρειαστεί να χρησιμοποιήσουν μετά όλα τα νήματα μέσα στη διαταγή διαμοίρασης για την εκτέλεση του δικού τους μέρους υπολογισμών), θα πρέπει πριν τη διαταγή διαμοίρασης καθ' αυτή να εισάγουμε και μία διαταγή *barrier*, όπως φαίνεται στο παράδειγμα που ακολουθεί.

```
#pragma omp parallel
 {
   Do_Job1();
   #pragma omp barrier
   #pragma omp for schedule(static)
   {
     Do_Job2();
   }
   Do_Job3();
 }
```

Άλλες περιπτώσεις συχνής χρήσης της διαταγής barrier σε παράλληλα προγράμματα OpenMP αφορούν, είτε

- σε σημεία του προγράμματος μετά το πέρας μίας διαταγής διαμοίρασης (και άλλων πιθανά εργασιών), όταν έχουμε χρησιμοποιήσει για κάποιο λόγο μέσα στη διαταγή διαμοίρασης τη δήλωση nowait, οπότε αίρεται ο συγχρονισμός φράγματος που εισάγει το ίδιο το OpenMP στο τέλος της διαταγής, είτε
- σε προγράμματα που επιτελούν πλήθος επαναλήψεων οι οποίες απαιτείται να είναι συγχρονισμένες (να μην αρχίζει δηλαδή η επόμενη επανάληψη για κανένα νήμα πριν να έχει ολοκληρωθεί η προηγούμενη για όλα τα νήματα) και

αποτελούνται από μίξη διαταγών διαμοίρασης εργασιών OpenMP και άλλου κώδικα εντός μιας παράλληλης περιοχής, ή

- σε περιπτώσεις που λόγω της φύσης του προβλήματος εφαρμόζουμε σε διάφορα σημεία (στα οποία δεν είναι δυνατό να αξιοποιήσουμε τις διαταγές και μηχανισμούς υψηλού επιπέδου που μας παρέχει το OpenMP) παραλληλισμό χαμηλότερου επιπέδου (όπως π.χ. όταν προγραμματίζουμε σε *posix threads*), οπότε εκ των πραγμάτων όπου απαιτείται συγχρονισμός φράγματος πρέπει να τον εισάγει ο ίδιος ο προγραμματιστής.

6.2.4.5 Η διαταγή *master*

Τέλος, πέραν της διαταγής διαμοίρασης εργασίας *single* την οποία είδαμε στην προηγούμενη παράγραφο (6.2.3), το OpenMP παρέχει έναν ακόμα πιο απλό τρόπο (με ισοδύναμη ως επί το πλείστον λειτουργικότητα) για τις περιπτώσεις που επιθυμούμε να περιορίσουμε σε κάποιο σημείο του προγράμματος, μέσα σε μία παράλληλη περιοχή, ένα κομμάτι κώδικα να εκτελεστεί από ένα μόνο νήμα (ως αν ήταν δηλαδή σε περιβάλλον ακολουθιακής εκτέλεσης), τη διαταγή *master*.

Η σύνταξη της διαταγής master έχει ως ακολούθως,

```
#pragma omp master new-line
      structured-block
```

και είναι ισοδύναμη με το ακόλουθο απόσπασμα κώδικα:

```
tid = omp_get_thread_num();
   if (tid == 0)
      structured-block
```

Οι διαφορές της δε από τη διαταγή *single* είναι συνοπτικά οι ακόλουθες:

- Με τη διαταγή *master* η εκτέλεση του σώματος εντολών εντός της διαταγής ανατίθεται πάντα στο νήμα-αφέντη (δηλαδή στο νήμα με αριθμό/ταυτότητα 0), ενώ με τη διαταγή *single* μπορεί να ανατεθεί σε οποιοδήποτε νήμα (στο πρώτο νήμα που θα βρεθεί διαθέσιμο).
- Στο τέλος της διαταγής *single* εφαρμόζεται συγχρονισμός φράγματος (*barrier* / ο οποίος μπορεί να αρθεί με την πρόσθετη χρήση της δήλωσης *nowait*), ενώ με τη διαταγή *master* τα υπόλοιπα νήματα (πλην του νήματος-αφέντη) συνεχίζουν σε

κάθε περίπτωση την εκτέλεσή τους κανονικά χωρίς να περιμένουν την ολοκλήρωση των εργασιών της διαταγής από το νήμα-αφέντη.

- Με τη διαταγή *single* ο προγραμματιστής έχει τη δυνατότητα να χρησιμοποιήσει και κάποιες επιπλέον (μικρής σημασίας ωστόσο) δυνατότητες (μέσω των διατιθέμενων για χρήση σε αυτήν δηλώσεων/παραμέτρων *private*, *firstprivate* και *copyprivate* – βλ. παρ. 2.6.3)

Η χρήση/λειτουργία της διαταγής *master* είναι γενικά πιο απλή και γρήγορη, η επιλογή ωστόσο κάθε φορά για το ποια από τις δύο διαταγές είναι προτιμότερη, εξαρτάται από τη στάθμιση των παραπάνω (έστω και μικρών) πλεονεκτημάτων/ μειονεκτημάτων, ανάλογα και με το είδος της εφαρμογής καθ' αυτής. Υπενθυμίζεται επίσης (όπως τονίστηκε και στην παρ. 6.2.3) ότι η διαταγή *single* μπορεί υπό συνθήκες να χρησιμοποιηθεί και για την υποστήριξη μη δομημένου παραλληλισμού (έστω και με μη ιδιαίτερα αποδοτικό τρόπο) [AC09]. Σε σχέση με τη λειτουργικότητά της αυτή δεν μπορεί να υποκατασταθεί από τη διαταγή master.

6.2.4.5 Η διαταγή *flush*

Μέσω της διαταγής *flush* μπορούμε να καθορίσουμε ένα «οριζόντιο» σημείο εκτέλεσης του συνόλου των νημάτων, στο οποίο θέλουμε να διασφαλιστεί ότι όλα τα νήματα έχουν συνεπή εικόνα συγκεκριμένων κοινών αντικειμένων στη μνήμη. Αυτό σημαίνει ότι τυχόν προηγούμενοι υπολογισμοί εκφράσεων οι οποίοι αναφέρονται στα προς συγχρονισμό αντικείμενα θα πρέπει να έχουν ολοκληρωθεί, και οι υπολογισμοί οι οποίοι έπονται πάνω σε αυτά θα πρέπει να μην έχουν ξεκινήσει ακόμα. Η σύνταξη της διαταγής flush έχει ως ακολούθως:

```
#pragma omp flush [list ...] new-line
```

Εάν όλα τα αντικείμενα για τα οποία απαιτείται συγχρονισμός μπορούν να καθοριστούν από μεταβλητές τότε αυτές δηλώνονται στην παράμετρο *list*. Εάν απουσιάζει η παράμετρος αυτή, τότε η διαταγή συγχρονίζει όλα τα κοινά αντικείμενα του προγράμματος. Μια τέτοια διαταγή (χωρίς παράμετρο *list*) υπονοείται εξ' ορισμού (με ευθύνη δηλαδή του ίδιου του λογισμικού) τόσο στα σημεία όπου έχουμε εισάγει οι ίδιοι μια διαταγή συγχρονισμού φράγματος (*barrier)* όσο και στα σημεία εξόδου των διαταγών *parallel*, *for*, *sections* και *single* (εκτός αν έχει δηλωθεί στις τελευταίες η παράμετρος *nowait*). Γενικότερα, η διαταγή *flush* μπορεί να θεωρηθεί ως μία πιο εξειδικευμένη διαταγή συγχρονισμού τύπου φράγματος, για συγκεκριμένα ωστόσο

μόνο κοινά αντικείμενα (ένα ή περισσότερα ή/και όλα) κατ' επιλογή του χρήστη/ προγραμματιστή και ανάλογα με τις ανάγκες της εφαρμογής του. Για περισσότερες λεπτομέρειες όσον αφορά τη χρήση της καθώς και για ενδεικτικά παραδείγματα μπορεί κανείς να ανατρέξει στα [OMP01,OMP02].

6.2.5 Ορισμός Διακριτών Εργασιών (tasks)

Στις αρχικές/βασικές του εκδόσεις το OpenMP (έως και την έκδοση 2.5) εστιάζει κατά κύριο λόγο στην αποδοτική παραλληλοποίηση καλά ορισμένων δομών (π.χ. δομημένων βρόγχων με προκαθορισμένο αριθμό επαναλήψεων), καθώς και στις ανάγκες υποστήριξης του απαιτούμενου συγχρονισμού μεταξύ των πολλαπλών νημάτων, χωρίς να εμβαθύνει ιδιαίτερα σε ειδικότερες μορφές αλγορίθμων με υψηλότερες απαιτήσεις όσον αφορά την παραλληλοποίησή τους, όπως π.χ. μη αυστηρά δομημένους βρόγχους (π.χ. loops με σύνθετες συνθήκες τερματισμού), δομές δεικτών (pointers), αναδρομικές διατυπώσεις αλγορίθμων κλπ.

Η ίδια η διαδικασία παραλληλοποίησης επίσης όταν συναντώνται παράλληλες περιοχές και διαταγές διαμοιρασμού εργασιών είναι αρκετά αυστηρά δομημένη. Δημιουργείται μία ομάδα από προκαθορισμένο αριθμό νημάτων, και στη συνέχεια σε κάθε νήμα (ή σε κάποια μόνο αυτά) ανατίθεται ένα συγκεκριμένο υποσύνολο εργασιών τις οποίες θα πρέπει να εκτελέσει μέχρι τέλους. Παρέχονται μεν κάποιες δυνατότητες δυναμικής διαχείρισης και ανάθεσης εργασιών (π.χ. δυναμική ανάθεση επαναλήψεων σε μία διαταγή διαμοίρασης *for*, δυνατότητα ορισμού εκτέλεσης ενός σώματος εργασίας από το πρώτο διαθέσιμο νήμα του συστήματος μέσω κατάλληλης χρήσης της διαταγής *single* - ή/και της διαταγής *sections* - με παράμετρο *nowait* κ.α.) δεν επαρκούν ωστόσο για την αποδοτική υλοποίηση σημαντικού φάσματος παράλληλων αλγορίθμων με μη εμφανή (μη δομημένη) μορφή παραλληλοποίησης (όπως οι μορφές που αναφέρθηκαν παραπάνω).

Ήταν εμφανές ότι θα αποτελούσε μία πολύ χρήσιμη (και θεμελιώδη όσον αφορά την ευρύτερη χρήση και αποδοχή του OpenMP) επέκταση, αν δινόταν η δυνατότητα δυναμικού ορισμού διακριτών εργασιών (tasks), οι οποίες να μπορούν να γεννηθούν και να αντιμετωπιστούν αυτόνομα κατά το χρόνο εκτέλεσης, ανάλογα με τις ανάγκες εκτέλεσης του αλγορίθμου, και όχι με αυστηρά εκ των προτέρων δομημένο τρόπο. Η επέκταση αυτή ολοκληρώθηκε και ενσωματώθηκε στην έκδοση 3.0 του OpenMP, μέσω της διαταγής *task* η οποία συντάσσεται ως ακολούθως:

```
#pragma omp task [clause ...] new-line
```

```
structured-block
```

όπου στο *clause* μπορούμε να ορίσουμε/δηλώσουμε τις ακόλουθες παραμέτρους:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default (shared | none)
```

Οι δηλώσεις/παράμετροι *if, private, shared, firstprivate* και *default* έχουν παρόμοια έννοια με αυτήν που αναφέρθηκε στις προηγούμενες παραγράφους για τις αντίστοιχες παραμέτρους της διαταγής *parallel* και των διαταγών διαμοίρασης εργασίας. Μέσω της δήλωσης/παραμέτρου *untied* ο προγραμματιστής μπορεί να ορίσει για μία διακριτή εργασία, αν κάποια στιγμή ανασταλεί η εκτέλεσή της από ένα νήμα, να υπάρχει η δυνατότητα να συνεχιστεί αυτή όταν θα είναι δυνατό από οποιοδήποτε νήμα της ομάδας (και όχι από το ίδιο νήμα στο οποίο εκτελείτο). Για περισσότερες λεπτομέρειες μπορεί κανείς να ανατρέξει στο [AC09].

Όταν κατά την εκτέλεση ενός νήματος μέσα σε μία παράλληλη περιοχή συναντάται η διαταγή *task*, δημιουργείται μία νέα διακριτή εργασία με δικό της κώδικα (αυτόν που ακολουθεί στο σώμα της διαταγής) και χώρο δεδομένων, η οποία θα είναι υποψήφια στη συνέχεια να ανατεθεί σε κάποιο διαθέσιμο νήμα για εκτέλεση. Το σύνολο των διακριτών εργασιών διαχωρίζονται πλέον ουσιαστικά σε δύο κατηγορίες: (α) αυτές που έχουν προκύψει από τις συμβατικές διαταγές (*parallel*, *for*, *sections*, *single* – implicit tasks), και (β) αυτές που έχουν προκύψει από τη διαταγή *task* (explicit tasks). Οι δεύτερες από τις παραπάνω είναι σαν να εισέρχονται με τη γέννησή τους σε ένα σωρό (pool) από τον οποίον στη συνέχεια θα επιλεγούν για εκτέλεση από τα νήματα που θα καθίστανται διαθέσιμα/ελεύθερα.

Προκειμένου να δούμε με ένα απλό παράδειγμα τη λειτουργικότητα του μηχανισμού των διακριτών εργασιών, ας θεωρήσουμε ότι θέλουμε να επεξεργαστούμε τα στοιχεία μίας δυναμικής λίστας στοιχείων. Για τέτοιου είδους λειτουργίες είναι μεν δυνατή η εφαρμογή των αυτοματοποιημένων μεθόδων και μηχανισμών του OpenMP που έχουμε δει μέχρι τώρα στα πλαίσια της παρούσας ενότητας, χωρίς ωστόσο να καθίσταται η χρήση τους επαρκώς αποδοτική. Ακολουθεί ενδεικτικά παρακάτω ένα απόσπασμα παράλληλου προγράμματος σε OpenMP, το οποίο θα μπορούσε να χρησιμοποιηθεί για τη διάτρεξη μιας δυναμικής λίστας με σκοπό να επιτελεστεί

κάποια επεξεργασία για κάθε στοιχείο της (μέσω της συνάρτησης *process()*), κάνοντας χρήση της διαταγής διαμοίρασης εργασίας *for*.

```
#pragma omp parallel default(shared)
 {
  #pragma omp single private(p)
   {
    p = headlist;
    num = 0;
    while (p)
     {
      array[num] = p;
      p = next(p);
      num++;
     }
   }
  #pragma omp for private(i)
   for (i=0; i<num; i++)
     process(array[i]);
 }
```

Όπως φαίνεται παραπάνω, για να είναι εφικτή η κατά ένα τρόπο τουλάχιστον παράλληλη επεξεργασία των στοιχείων της λίστας, καθίσταται απαραίτητη η αρχική διάτρεξη της λίστας προκειμένου να βρεθούν και να αποθηκευτούν ανεξάρτητα οι δείκτες όλων των κόμβων/στοιχείων αυτής. Αφού ολοκληρωθεί η αρχική αυτή διάτρεξη (και καθοριστεί μεταξύ άλλων και ο συνολικός αριθμός των κόμβων της λίστας), μέσω της διαταγής *for* διαμοιράζεται κατάλληλα ο φόρτος επεξεργασίας των στοιχείων της λίστας σε όλα τα νήματα. Η διαδικασία της αρχικής διάτρεξης είναι εμφανώς πολύ δύσκολο να παραλληλοποιηθεί (με τους υφιστάμενους μηχανισμούς – λόγω της δυναμικής διασύνδεσης δεικτών) και για το λόγο αυτό την ενσωματώνουμε ενδεικτικά σε μία διαταγή single με σκοπό την ορθή ακολουθιακή εκτέλεσή της από ένα μόνο νήμα (θα μπορούσαμε ισοδύναμα να την εκτελέσουμε και εκτός της παράλληλης περιοχής). Έτσι παραλληλοποιείται μεν αποδοτικά ένα (το βασικό) μέρος της όλης διαδικασίας (αυτό της τελικής επεξεργασίας), θα πρέπει ωστόσο να αποδεχθούμε την επιβάρυνση που προκύπτει από την ανάγκη της αρχικής διάτρεξης της λίστας κατά την εκκίνηση του αλγορίθμου, η οποία είναι ανάλογη του μήκους της λίστας.

Με χρήση της διαταγής *task* θα μπορούσε αντίθετα να γραφτεί το ανωτέρω πρόγραμμα πολύ πιο ευέλικτα και αποδοτικά, όπως φαίνεται παρακάτω:

```
#pragma omp parallel default(shared)
 {
  #pragma omp single private(p)
```

```
    {
     p = headlist;
     while (p)
      {
       #pragma omp task
         process(p);
       p=next(p);
      }
    }
 }
```

Βασική πτυχή της ανωτέρω διατύπωσης αποτελεί το γεγονός ότι με τη διαταγή *task* μας δίνεται η δυνατότητα με ευθύ/άμεσο τρόπο να εκκινήσουμε μία διακριτή εργασία για την επεξεργασία κάθε νέου κόμβου που συναντάμε, εκείνη ακριβώς τη στιγμή που τον συναντάμε. Η σχετική δε επαναληπτική διαδικασία γέννησης διακριτών εργασιών ενσωματώνεται μέσα σε μία διαταγή *single* προκειμένου να διασφαλιστεί η εκτέλεσή της από ένα μόνο νήμα (το πρώτο διαθέσιμο εκείνη τη χρονική στιγμή). Τα υπόλοιπα νήματα θα λαμβάνουν προοδευτικά εργασίες προς εκτέλεση από αυτές που γεννιούνται, κάτι το οποίο θα συμβεί και για το νήμα που θα εκτελέσει τον κώδικα της διαταγής *single* όταν τελειώσει με αυτόν.

Θα πρέπει να σημειωθεί εδώ ότι η παραπάνω διαδικασία παράλληλης διάτρεξης και επεξεργασίας των κόμβων μίας δυναμικής λίστας, θα μπορούσε να γραφτεί εναλλακτικά (χωρίς τη χρήση της διαταγής *task*) και με έναν ακόμα πλάγιο αλλά όχι επαρκώς αποδοτικό τρόπο, και πιο συγκεκριμένα με κατάλληλη χρήση της διαταγής *single* και παράμετρο *nowait*, όπως περιγράφεται αναλυτικότερα στο [AC09].

Ιδιαίτερη σημασία κατά την ανάπτυξη και προτυποποίηση του μηχανισμού των διακριτών εργασιών δόθηκε επίσης στις δυνατότητες συγχρονισμού που θα έπρεπε να τον συνοδεύουν, προκειμένου να μπορεί να υποστηριχθεί αποδοτικά η διατύπωση σύνθετων αλγορίθμων με πολλαπλές εξαρτήσεις ή καθοδηγούμενες αναδρομικές κλήσεις. Ειδικότερα, στις νέες εκδόσεις του OpenMP, κάθε διακριτή εργασία με τη γέννησή της αποτελεί πλέον αυτομάτως *παιδί* του νήματος/εργασίας από την οποία γεννήθηκε. Έτσι είναι εύκολα δυνατόν να υποστηριχθούν και αντίστοιχες δομές συγχρονισμού, όπως π.χ. αναμονής μίας διακριτής εργασίας για την ολοκλήρωση μίας άλλης κλπ. Η παραπάνω δυνατότητα παρέχεται πιο συγκεκριμένα μέσω της διαταγής *taskwait* η οποία συντάσσεται ως ακολούθως:

```
#pragma omp taskwait newline
```

Όταν συναντηθεί μία διαταγή *taskwait* αναστέλλεται η εκτέλεση της τρέχουσας διακριτής εργασίας μέχρι όλα τα παιδιά της ολοκληρώσουν τη δικιά τους

διακριτή εργασία. Η αναμονή ολοκλήρωσης αυτή ελέγχεται μόνο για του πρώτου επιπέδου (άμεσα) παιδιά, όχι για βαθύτερου επιπέδου (εγγόνια κλπ). Ένα ενδεικτικό παράδειγμα χρήσης της διαταγής *taskwait* δίνεται παρακάτω για τη διάτρεξη και επεξεργασία των στοιχείων των κόμβων ενός δυαδικού δέντρου σε διάταξη *πρώτα-σε-βάθος* και *postorder* (πρώτα δηλαδή επίσκεψη και επεξεργασία των κόμβων των παιδιών και μετά του κόμβου-ρίζας).

```
void postorder(node)
 {
  if (node->left)
   #pragma omp task
     postorder(node->left);

  if (node->right)
   #pragma omp task
     postorder(node->right);

  #pragma omp taskwait

  process(node->data);
 }
```

Το παράδειγμα αυτό είναι ιδιαίτερα διδακτικό για τον επιπλέον λόγο ότι επιδεικνύει μεταξύ των άλλων, πώς μπορεί να διατυπωθεί ευέλικτα και αποδοτικά ένας αναδρομικός αλγόριθμος σε περιβάλλον OpenMP με χρήση της διαταγής *task*. Είναι επίσης εμφανές στο εν λόγω παράδειγμα ότι δεν αρκεί μόνο η δυναμική (και αναδρομικά εκφραζόμενη) γέννηση των απαιτούμενων διακριτών εργασιών κατά τον τρόπο που περιγράψαμε μέχρι τώρα, αλλά απαιτείται και ο καθορισμός κάποιων υποδηλούμενων προτεραιοτήτων οι οποίες θα πρέπει να τηρηθούν απαραίτητα κατά την εκτέλεσή τους. Η ανάγκη αυτή καθορισμού προτεραιοτήτων καλύπτεται κατάλληλα μέσω της διαταγής συγχρονισμού *taskwait*.

6.2.6 Άλλες Δυνατότητες & Παραδείγματα

6.2.6.1 Η δήλωση *reduction*

Μία πολύ συχνή ανάγκη κατά την υλοποίηση παράλληλων αλγορίθμων αποτελεί η συγκέντρωση επιμέρους αποτελεσμάτων από κάθε νήμα/επεξεργαστή που είναι αποθηκευμένα σε μία ή περισσότερες μεταβλητές, και η εφαρμογή μίας πράξης σε αυτά (π.χ. πρόσθεση, πολλαπλασιασμός, εύρεση ελαχίστου/μεγίστου κλπ), μια λειτουργία η οποία μας είναι γνωστή ως *reduction*. Το OpenMP παρέχει έναν

αυτοματοποιημένο μηχανισμό υλοποίησης της λειτουργίας του *reduction*, μέσω αντίστοιχης δήλωσης/παραμέτρου, η οποία μπορεί να συμπεριληφθεί είτε σε μία διαταγή παραλληλισμού (*parallel*) είτε σε μία από τις δύο πρώτες διαταγές διαμοιρασμού εργασίας (*for, sections*) και έχει την ακόλουθη μορφή:

```
reduction (operator: list)
```

Ως παράδειγμα, έστω ότι μας ζητείται να υπολογίσουμε το εσωτερικό γινόμενο δύο διανυσμάτων *a* και *b* (μήκους *N*) και να αποθηκεύσουμε το τελικό αποτέλεσμα στην κοινή μεταβλητή *result.* Η αντίστοιχη επαναληπτική δομή την οποία θα χρησιμοποιούσαμε για την ακολουθιακή υλοποίηση σε C, θα ήταν της μορφής:

```
for (i=0; i < N; i++)
   {
      result = result + (a[i] * b[i]);
   }
```

Για τον παράλληλο υπολογισμό των παραπάνω επαναλήψεων, είναι εύκολο να παρατηρήσουμε ότι απαιτούνται τα ακόλουθα επιμέρους βήματα: (α) ο διαμοιρασμός των τιμών των διανυσμάτων στα δημιουργηθέντα νήματα, (β) ο πολλαπλασιασμός από κάθε νήμα τιμή προς τιμή των τιμών που του αναλογούν από κάθε διάνυσμα και (γ) η πρόσθεση όλων των όρων που προέκυψαν σε μία κοινή μεταβλητή που αναπαριστά και το τελικό αποτέλεσμα. Από τα ανωτέρω βήματα τα (α) και (β) μπορούν να υλοποιηθούν παράλληλα με αποδοτικό τρόπο (όπως έχουμε ήδη δει για παρόμοια εργασία - πρόσθεση των τιμών δύο διανυσμάτων - σε προηγούμενη παράγραφο/6.2.3) με κατάλληλη εφαρμογή μίας διαταγής διαμοιρασμού εργασίας *for*.

Όσον αφορά το βήμα (γ) ένα μέρος αυτού μπορεί και αυτό μεν να υλοποιηθεί παράλληλα με παρόμοιο τρόπο (η πρόσθεση των επιμέρους όρων που έχει να υπολογίσει το κάθε νήμα για τα κομμάτια των δύο διανυσμάτων που του αναλογούν), έχει την ιδιαιτερότητα ωστόσο ότι για να ολοκληρωθεί ο υπολογισμός θα πρέπει να πραγματοποιηθεί επιπρόσθετα (με επίσης παράλληλο τρόπο) και μία συνολική πρόσθεση στη συνέχεια όλων των επιμέρους αθροισμάτων που υπολόγισαν τα νήματα. Η εκτέλεση της πρόσθεσης αυτής θα πρέπει επιπλέον να προστατευτεί κατάλληλα (διασφάλιση αμοιβαίου αποκλεισμού) καθώς θα πρέπει να γίνει επί μίας κοινής/ σφαιρικής μεταβλητής, η ταυτόχρονη αλλαγή της οποίας από περισσότερα του ενός νήματα αποτελεί όπως ξέρουμε κρίσιμη περιοχή.

Αυτής της μορφής την ολοκληρωμένη παράλληλη υλοποίηση που υπονοεί το βήμα (γ) του παραπάνω υπολογισμού (συγκέντρωση όλων των επιμέρους τιμών και

εφαρμογή μίας πράξης λαμβάνοντας επίσης πρόνοια για αμοιβαίο αποκλεισμό όπου συναντάται κρίσιμη περιοχή) αναλαμβάνει να μας προσφέρει αυτοματοποιημένα η δήλωση reduction του OpenMP. Η ολοκληρωμένη παράλληλη υλοποίηση του ανωτέρω παραδείγματος με χρήση της δήλωσης *reduction*, παρατίθεται παρακάτω:

```
#include <omp.h>
#define CHUNKSIZE 10
#define N     100

main ()  {

int   i, chunk;
float a[N], b[N], result;

/* Αρχικοποιήσεις */
chunk = CHUNKSIZE;
result = 0.0;
for (i=0; i < N; i++)
  {
  a[i] = 1.0;
  b[i] = 2.0;
  }

#pragma omp parallel default(shared) private(i)
{
#pragma omp for schedule(static,chunk) reduction(+:result)
  for (i=0; i < N; i++)
    {
       result = result + (a[i] * b[i]);
    }
}
printf("Final result= %f\n", result);
}
```

Όταν συναντάται στα πλαίσια της παραπάνω διαταγής διαμοιρασμού *for* η δήλωση *reduction*, ορίζεται πιο συγκεκριμένα για κάθε νήμα ένα ιδιωτικό αντίγραφο της μεταβλητής αυτής που δηλώνουμε στην παρένθεση (*result*), δίνοντας έτσι τη δυνατότητα σε κάθε νήμα να χρησιμοποιήσει τη μεταβλητή αυτή απρόσκοπτα τοπικά για την τοπική συγκέντρωση και πράξη αναφορικά με τις επαναλήψεις που του αναλογούν. Στο τέλος (αφού ολοκληρώσουν τις τοπικές πράξεις/αθροίσεις τους όλα τα νήματα) πραγματοποιεί παράλληλα την πρόσθεση όλων σε μία (κοινή/σφαιρική πλέον) μεταβλητή με το ίδιο όνομα, καταργώντας όλα τα ιδιωτικά αντίγραφα αυτής που είχε δημιουργήσει. Η κοινή/σφαιρική αυτή μεταβλητή περιέχει και το τελικό αποτέλεσμα.

Εάν δηλώσουμε στην παρένθεση του *reduction* (παράμετρος *list*) παραπάνω από μία μεταβλητές, θα πραγματοποιηθεί ή ίδια λειτουργία, ξεχωριστά για κάθε μία

από τις δηλωθείσες μεταβλητές της λίστας. Οι πράξεις δε (παράμετρος *operator*) οι οποίες επιτρέπονται να πραγματοποιηθούν σε μία λειτουργία *reduction* στην έκδοση OpenMP 3.0 είναι οι ακόλουθες: +, *, -, &, ^, |, &&, ||. Στην έκδοση 3.1 έχουν προστεθεί επίσης και οι ιδιαίτερα χρήσιμοι τελεστές *min*/*max* (υποστηρίζονταν μέχρι πρότινος μόνο για τη Fortran), ενώ στην έκδοση 4.0 θα δίνεται η δυνατότητα ορισμού αντίστοιχων πρόσθετων πράξεων και από το χρήστη (user defined).

Αν δεν είχαμε στη διάθεσή μας τον μηχανισμό του *reduction*, για να υλοποιήσουμε παράλληλα τον υπολογισμό του παραπάνω παραδείγματος (εσωτερικό γινόμενο δύο διανυσμάτων) με ισοδύναμο τρόπο, θα έπρεπε (α) να ορίσουμε (και δηλώσουμε ως ιδιωτική/private) από μόνοι μας μία ξεχωριστή μεταβλητή (π.χ. *loc_result*) στην οποία να πραγματοποιείται από κάθε νήμα τοπικά (μέσα στη σχετική διαταγή *for*) η άθροιση των επιμέρους όρων (γινομένων) που έχει υπολογίσει, και (β) να αθροίσουμε στη συνέχεια την τελική τιμή της ιδιωτικής αυτής μεταβλητής, για κάθε νήμα, στην κοινή/σφαιρική μεταβλητή *result* σε ξεχωριστή εντολή έξω από τη διαταγή *for*. Η τελική αυτή άθροιση θα πρέπει να προστατευτεί ως κρίσιμη περιοχή με χρήση κάποιου από τους αντίστοιχους μηχανισμούς του OpenMP (*critical*, *atomic* ή *locks*). Ο σχετικός κώδικας θα είχε ενδεικτικά την ακόλουθη μορφή:

```
#include <omp.h>
#define CHUNKSIZE 10
#define N     100

main ()  {

int   i, chunk;
float a[N], b[N], loc_result, result;

/* Αρχικοποιήσεις */
chunk = CHUNKSIZE;
result = 0.0;
for (i=0; i < N; i++)
  {
  a[i] = 1.0;
  b[i] = 2.0;
  }

#pragma omp parallel shared(result) private(i,loc_result)
  {
    loc_result = 0.0;
    #pragma omp for schedule(static,chunk)
       for (i=0; i < N; i++)
         {
            loc_result = loc_result + (a[i] * b[i]);
         }
```

```
    #pragma omp atomic
       result += loc_result;
  }

printf("Final result= %f\n", result);
}
```

Από τα παραπάνω, είναι εύκολο να διαπιστώσει κανείς ότι με χρήση του μηχανισμού του *reduction*, επιτυγχάνεται το ζητούμενο αποτέλεσμα, πιο κομψά, οικονομικά και ευέλικτα και με εγγυημένα βέλτιστο τρόπο, φροντίζοντας παράλληλα για την διασφάλιση του αμοιβαίου αποκλεισμού (όπου αυτός απαιτείται) με διάφανο τρόπο, χωρίς δηλαδή να απασχολεί το χρήστη η υλοποίησή του.

6.2.6.2 Η δήλωση *collapse*

Σε μία διαταγή διαμοιρασμού εργασίας *for*, παρέχεται από το OpenMP επίσης η δυνατότητα ενιαίου διαμοιρασμού ενός συνόλου επαναλήψεων οι οποίες δηλώνονται στα πλαίσια όχι μόνο ενός (όπως στα παραδείγματα που είδαμε στην παράγραφο 6.2.3) αλλά και περισσοτέρων του ενός εμφωλιασμένων (το ένα μέσα στο άλλο) *for-loops*, διευκολύνοντας έτσι τον αρτιότερο δυνατό διαμοιρασμό φόρτου και δεδομένων σε πιο σύνθετα προβλήματα και υπολογισμούς. Η δυνατότητα αυτή παρέχεται μέσω της δήλωσης *collapse*, η οποία συντάσσεται ως ακολούθως:

```
collapse (n)
```

όπου με την παράμετρο *n* ορίζεται ο αριθμός των επιπέδων εμφωλιασμού που επιθυμούμε, δηλαδή μέχρι πόσα στη σειρά εμφωλιασμένα *for-loops* θέλουμε να ενοποιηθούν και να διαμοιραστούν οι επαναλήψεις ως αν ήταν ένα. Ως παράδειγμα ας δούμε πως μπορεί να εφαρμοστεί η παραπάνω δήλωση στο πρόβλημα του πολλαπλασιασμού δύο δυσδιάστατων πινάκων. Υποθέτοντας ότι μας ζητείται να πολλαπλασιάσουμε δύο δυσδιάστατους πίνακες a, b (για τους οποίους ας θεωρήσουμε για ευκολία και χωρίς βλάβη της γενικότητας ότι είναι τετραγωνικοί, δηλ. διαστάσεων NxN) και να αποθηκεύσουμε το αποτέλεσμα στον πίνακα c, η αντίστοιχη επαναληπτική δομή την οποία θα χρησιμοποιούσαμε για την ακολουθιακή υλοποίηση σε C, θα ήταν της παρακάτω μορφής:

```
for (i=0; i<N; i++)
   for(j=0; j<N; j++)
     for (k=0; k<N; k++)
       c[i][j] += a[i][k] * b[k][j];
```

Εξετάζοντας διεξοδικά τους δυνατούς τρόπους παραλληλισμού του παραπάνω αποσπάσματος, με βάση τις επιπλέον επιλογές που μας προσφέρει η δήλωση/ μηχανισμός *collapse*, μπορούμε να παρατηρήσουμε τα εξής:

- Αν παραλληλοποιήσουμε το παραπάνω σύνολο βρόγχων με εμφωλιασμό ενός μόνο επιπέδου (collapse(1)), το σύνολο των επαναλήψεων που θα διαμοιραστούν ενιαία θα είναι N (θα γίνει δηλαδή διαμοιρασμός των απαιτούμενων υπολογισμών ανά γραμμές του πίνακα αποτελέσματος), και για κάθε μία από αυτές κάθε νήμα που την λαμβάνει θα εκτελεί ότι υπολογισμούς έπονται για αυτήν στα πλαίσια των επόμενων δύο βρόγχων. Η μορφή αυτή διαμοιρασμού αποτελεί και την εξ' ορισμού μορφή διαμοιρασμού η οποία θα ακολουθείτο από το OpenMP αν δεν υπήρχε καθόλου η δήλωση *collapse*.
- Αν παραλληλοποιήσουμε το παραπάνω σύνολο βρόγχων με εμφωλιασμό δύο επιπέδων (collapse(2)), το σύνολο των επαναλήψεων που θα διαμοιραστούν ενιαία θα είναι πλέον N^2 (θα γίνει δηλαδή διαμοιρασμός των απαιτούμενων υπολογισμών ανά στοιχείο του πίνακα αποτελέσματος), και για κάθε μία από αυτές κάθε νήμα που την λαμβάνει θα εκτελεί ότι υπολογισμούς έπονται για αυτήν στα πλαίσια του τρίτου βρόγχου.
- Αν παραλληλοποιήσουμε το παραπάνω σύνολο βρόγχων με εμφωλιασμό τριών επιπέδων (collapse(3)) το σύνολο των επαναλήψεων που θα διαμοιραστούν ενιαία θα είναι N^3 (θα γίνει δηλαδή διαμοιρασμός των απαιτούμενων υπολογισμών ανά όρο του κάθε στοιχείου του πίνακα αποτελέσματος), και για κάθε μία από αυτές κάθε νήμα που την λαμβάνει θα εκτελεί το αντίστοιχο ζεύγος πολλαπλασιασμού και πρόσθεσης, όπως αυτά καθορίζονται στα πλαίσια του τρίτου βρόγχου. Στην τελευταία περίπτωση, θα πρέπει να προσέξουμε ότι η πράξη της πρόσθεσης στο εσωτερικό του τρίτου βρόγχου θα πρέπει να προστατευτεί ως κρίσιμη περιοχή (με χρήση της διαταγής *critical* ή της *atomic* ή με χρήση μηχανισμού κλειδώματος του OpenMP), καθώς γίνεται από κάθε νήμα πάνω σε κάποια κοινή/σφαιρική μεταβλητή (κάποιο στοιχείο του πίνακα αποτελέσματος) στην οποία μπορεί να γράφει παράλληλα και κάποιο άλλο νήμα.

Ακολουθεί παρακάτω ένα ολοκληρωμένο παράλληλο πρόγραμμα σε OpenMP που υλοποιεί τη δεύτερη από τις περιπτώσεις που εξετάσαμε παραπάνω. Πιο συγκεκριμένα στα πλαίσια της υλοποίησης αυτής θεωρούμε ότι η διάσταση των πινάκων είναι ίση με N=4 (4x4) και ότι η διαμοίραση γίνεται με εμφωλιασμό δύο επιπέδων (*collapse(2)*) και σε ομάδες των δύο επαναλήψεων (*chunk=2*).

```
#include <omp.h>
#define N  4

int main (int argc, char *argv[])
{
int   tid, nth, i, j, k, chunk;
double a[N][N],b[N][N],c[N][N];
chunk = 2;

#pragma omp parallel default(shared) private(tid,i,j,k)
  {
  tid = omp_get_thread_num();

  /* Αρχικοποιήσεις των πινάκων */

  #pragma omp for schedule (static, chunk)
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
       a[i][j]= i+j;

  #pragma omp for schedule (static, chunk)
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
       b[i][j]= i*j;

  #pragma omp for schedule (static, chunk)
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
       c[i][j]= 0;

  /* Εκτέλεση του πολλαπλασιασμού */

  #pragma omp for schedule (static, chunk) collapse (2)
  for (i=0; i<N; i++)
   for(j=0; j<N; j++)
    {
     printf("Thread %d computes iter %d,%d\n",tid,i,j);
     for (k=0; k<N; k++)
       c[i][j] += a[i][k] * b[k][j];
    }
  }

printf("*************************\n");
printf("Final Matrix:\n");
for (i=0; i<N; i++)
  {
  for (j=0; j<N; j++)
    printf("%5.2f  ", c[i][j]);
  printf("\n");
  }
printf("*************************\n");
}
```

Αν τρέξουμε το παραπάνω πρόγραμμα σε ένα περιβάλλον δύο επεξεργαστών/ πυρήνων (με δύο δηλαδή παράλληλα νήματα) θα παραχθεί η ακόλουθη έξοδος:

```
Thread 0 computes iter 0,0
Thread 0 computes iter 0,1
Thread 0 computes iter 1,0
Thread 0 computes iter 1,1
Thread 0 computes iter 2,0
Thread 0 computes iter 2,1
Thread 0 computes iter 3,0
Thread 0 computes iter 3,1
Thread 1 computes iter 0,2
Thread 1 computes iter 0,3
Thread 1 computes iter 1,2
Thread 1 computes iter 1,3
Thread 1 computes iter 2,2
Thread 1 computes iter 2,3
Thread 1 computes iter 3,2
Thread 1 computes iter 3,3
**************************
Final Matrix:
 0.00  14.00  28.00  42.00
 0.00  20.00  40.00  60.00
 0.00  26.00  52.00  78.00
 0.00  32.00  64.00  96.00
**************************
```

Το 1ο νήμα θα αναλάβει (από την πρώτη γραμμή) τις επαναλήψεις [0,0] και [0,1] και το 2ο νήμα τις επαναλήψεις [0,2] και [0,3], στη συνέχεια το 1ο νήμα θα αναλάβει (από την δεύτερη γραμμή πλέον) τις επαναλήψεις [1,0] και [1,1] και το 2ο νήμα τις επαναλήψεις [1,2] και [1,3], κοκ. Αντίθετα αν τρέξουμε το ίδιο πρόγραμμα με εμφωλιασμό ενός μόνο επιπέδου (collapse (1)) το 1ο νήμα θα αναλάβει όλες τις επαναλήψεις της πρώτης και της δεύτερης γραμμής και το 2ο νήμα θα αναλάβει όλες τις επαναλήψεις της τρίτης και της τέταρτης γραμμής. Η αντίστοιχη έξοδος που θα παραχθεί από το πρόγραμμα σε αυτήν την περίπτωση θα είναι η ακόλουθη:

```
Thread 0 computes iter 0,0
Thread 0 computes iter 0,1
Thread 0 computes iter 0,2
Thread 0 computes iter 0,3
Thread 0 computes iter 1,0
Thread 0 computes iter 1,1
```

```
Thread 0 computes iter 1,2
Thread 0 computes iter 1,3
Thread 1 computes iter 2,0
Thread 1 computes iter 2,1
Thread 1 computes iter 2,2
Thread 1 computes iter 2,3
Thread 1 computes iter 3,0
Thread 1 computes iter 3,1
Thread 1 computes iter 3,2
Thread 1 computes iter 3,3
**************************
Final Matrix:
 0.00  14.00  28.00  42.00
 0.00  20.00  40.00  60.00
 0.00  26.00  52.00  78.00
 0.00  32.00  64.00  96.00
**************************
```

Για να κατανοήσουμε καλύτερα δε, την επίτευξη αρτιότερης διαμοίρασης αν εφαρμόσουμε εμφωλιασμό δύο επιπέδων (*collapse(2)*) αντί για ένα (*collapse(1)*), ας υποθέσουμε ότι μας δίνονται πίνακες διαστάσεων 5x5 (N=5) και έχουμε στη διάθεσή μας πάλι δύο (2) παράλληλα νήματα (το N δηλαδή να μην είναι ακέραιο πολλαπλάσιο του αριθμού των νημάτων). Θεωρείστε επίσης ότι chunk=1 ώστε να μην επηρεάζεται πρόσθετα ή δυνατότητα άρτιας διαμοίρασης των επαναλήψεων. Σε αυτήν την περίπτωση αν χρησιμοποιήσουμε εμφωλιασμό ενός επιπέδου το 1^{ο} νήμα είναι εύκολο να διαπιστώσουμε ότι θα αναλάβει τις επαναλήψεις 3 γραμμών (συνολικά δηλαδή 3x5=15 στοιχείων) του πίνακα αποτελέσματος, ενώ το 2^{ο} νήμα θα αναλάβει τις επαναλήψεις 2 γραμμών (συνολικά δηλαδή 2x5=10 στοιχείων) του πίνακα αποτελέσματος. Αντίθετα αν χρησιμοποιήσουμε εμφωλιασμό δύο επιπέδων (ενιαίο δηλαδή διαμοιρασμό των 5x5=25 επαναλήψεων των δύο εξωτερικών βρόγχων), το 1^{ο} νήμα θα αναλάβει τις επαναλήψεις 13 συνολικά στοιχείων του πίνακα αποτελέσματος ενώ το 2^{ο} νήμα θα αναλάβει τις επαναλήψεις 12 στοιχείων.

6.2.6.3 Άλλες δυνατότητες εμφωλιασμού

Πέραν της δυνατότητας εμφωλιασμού παράλληλων διαταγών *for* μέσω της δήλωσης *collapse*, παρέχονται από το OpenMP και γενικότερες δυνατότητες εμφωλιασμένου παραλληλισμού μέσω των οποίων καθίσταται δυνατή η διατύπωση σε παράλληλο μορφή πιο σύνθετων αλγοριθμικών δομών. Πιο συγκεκριμένα, μέσα σε μία διαταγή *parallel* μπορεί κανείς να ορίσει/εμφωλιάσει άλλη διαταγή *parallel* κοκ ως ένα μέγιστο επιτρεπόμενο αριθμό επιπέδων ο οποίος ορίζεται από το σύστημα (βλ.

παράγραφο 6.2.6.5). Κάθε φορά που μέσα σε μία διαταγή *parallel* συναντάται μία άλλη διαταγή *parallel*, για κάθε νήμα που εκτελεί την πρώτη διαταγή εκκινείται μία νέα ομάδα νημάτων κάθε ένα από τα οποία εκτελεί τα περιεχόμενα της δεύτερης διαταγής. Ως αποτέλεσμα, αν εμφωλιάσουμε π.χ. *k* διαταγές parallel τη μία μέσα στην άλλη, και βρισκόμαστε σε ένα περιβάλλον με *t* επεξεργαστές/πυρήνες (οπότε ο αριθμός των νημάτων που θα δημιουργούνται για κάθε νέα ομάδα – αν δεν τον διαφοροποιήσουμε με κάποιο άλλο τρόπο – θα είναι επίσης *t*), θα δημιουργηθούν συνολικά t^k νήματα. Τα νήματα δε της κάθε νέας ομάδας θα αριθμούνται πάντα από *0* έως *t-1*, ως εσωτερικά νήματα δηλαδή της ομάδας τους, και όχι με κάποια καθολική αρίθμηση. Παρακάτω παρατίθεται ένα ενδεικτικό παράδειγμα με τρεις παράλληλες περιοχές (διαταγές parallel) εμφωλιασμένες η μία μέσα στην άλλη:

```
#include <omp.h>
main(int argc, char **argv)
 {
  int tid;
  #pragma omp parallel private(tid)
   {
    tid = omp_get_thread_num();
    printf("Hello 1 !!! from thread %d\n", tid);
    #pragma omp parallel private(tid)
     {
      tid = omp_get_thread_num();
      printf("Hello 2 !!! from thread %d\n", tid);
      #pragma omp parallel private(tid)
       {
        tid = omp_get_thread_num();
        printf("Hello 3 !!! from thread %d\n", tid);
       }
     }
   }
 }
```

Αν εκτελέσουμε τον παραπάνω κώδικα σε ένα υπολογιστικό σύστημα με δύο (2) επεξεργαστές/πυρήνες (οπότε θα δημιουργούνται δύο νήματα για κάθε νέα ομάδα), θα παραχθεί η ακόλουθη έξοδος (μέσα στην πρώτη παράλληλη περιοχή θα έχουν δημιουργηθεί και θα εκτυπώσουν το σχετικό διαγνωστικό μήνυμα δύο νήματα, στη δεύτερη περιοχή τέσσερα νήματα και στην τρίτη περιοχή οκτώ):

```
Hello 1 !!! from thread 0
Hello 1 !!! from thread 1
Hello 2 !!! from thread 0
Hello 2 !!! from thread 1
Hello 2 !!! from thread 0
```

```
Hello 2 !!! from thread 1
Hello 3 !!! from thread 0
Hello 3 !!! from thread 1
Hello 3 !!! from thread 0
Hello 3 !!! from thread 1
Hello 3 !!! from thread 0
Hello 3 !!! from thread 1
Hello 3 !!! from thread 0
Hello 3 !!! from thread 1
```

Προκειμένου να λειτουργήσει κατά τον παραπάνω τρόπο το OpenMP θα πρέπει η δυνατότητα εμφωλιασμένου παραλληλισμού να είναι ενεργοποιημένη. Αυτό μπορεί να επιτευχθεί θέτοντας εξωτερικά την μεταβλητή περιβάλλοντος OMP_NESTED ίση με *true* (ή εναλλακτικά χρησιμοποιώντας κατάλληλα τη συνάρτηση βιβλιοθήκης *omp_set_nested()* – βλ. παρ. 6.2.6.5). Θα πρέπει να σημειωθεί ότι η εν λόγω μεταβλητή περιβάλλοντος είναι στις περισσότερες υπάρχουσες υλοποιήσεις του OpenMP αρχικοποιημένη σε *false*. Σε μία τέτοια περίπτωση (αν δηλαδή η δυνατότητα εμφωλιασμένου παραλληλισμού είναι απενεργοποιημένη) κάθε φορά που μέσα σε μία διαταγή *parallel* συναντάται μία άλλη διαταγή *parallel*, για κάθε νήμα που εκτελεί την πρώτη διαταγή εκκινείται μεν τυπικά μία νέα ομάδα νημάτων η οποία ωστόσο αποτελείται μόνο από ένα (το τρέχον) νήμα. Ως αποτέλεσμα, η έξοδος του ενδεικτικού προγράμματος που παρουσιάσαμε παραπάνω σε μια τέτοια περίπτωση θα ήταν η ακόλουθη (τόσο μετά την πρώτη όσο και μετά τη δεύτερη όσο και μετά την τρίτη διαταγή τα ενεργά νήματα παραμένουν να είναι μόνο δύο):

```
Hello 1 !!! from thread 0
Hello 1 !!! from thread 1
Hello 2 !!! from thread 0
Hello 2 !!! from thread 0
Hello 3 !!! from thread 0
Hello 3 !!! from thread 0
```

Μια χαρακτηριστική εφαρμογή του εμφωλιασμένου παραλληλισμού αποτελεί η χρήση της διαταγής *sections* για την αναπαράσταση αναδρομικών αλγορίθμων, όπως φαίνεται π.χ. παρακάτω για έναν απλό αλγόριθμο διάτρεξης ενός δυαδικού δέντρου (ακολουθώντας την *πρώτα-σε-βάθος* / *postorder* διάταξη – όπως και στο παράδειγμα που είδαμε στην παράγγαρφο 6.2.5). Η μέθοδος αυτή (και γενικότερα η εξειδικευμένη χρήση τόσο της διαταγής *sections* όσο και της διαταγής *single* για τη διατύπωση μη δομημένου παραλληλισμού) αποτελούσε συνήθη πρακτική σε αντίστοιχες περιπτώσεις πριν ενσωματωθεί στο OpenMP ο μηχανισμός των διακριτών εργασιών, έχοντας

ωστόσο πολλά μειονεκτήματα όσον αφορά την τελική απόδοση των αλγορίθμων (για περισσότερες λεπτομέρειες βλ. [AC09]).

```
void traverse (node)
 {
   #pragma omp parallel shared(node)
      #pragma omp sections
        {
          #pragma omp section
             if (node->left)
               traverse (node->left)

          #pragma omp section
             if (node->right)
               traverse (node->left)
        }
   process(node->data)
 }
```

Αφού δημιουργηθεί η πρώτη παράλληλη περιοχή, η εκτέλεση κάθε αναδρομικής κλήσης (για το αριστερό ή το δεξί παιδί κάθε κόμβου) ανατίθεται μέσω της διαταγής *sections* σε ένα διαφορετικό νήμα. Το νήμα αυτό κατά την εκτέλεση της κλήσης θα δημιουργήσει μία νέα παράλληλη περιοχή (εμφωλιασμένα δηλαδή στην πρώτη) όπου μέσω μια νέας διαταγής *sections* θα αναθέσει τις νέες αναδρομικές κλήσεις για τα δικά του παιδιά σε δύο νέα νήματα κοκ. Όταν ολοκληρωθεί η εκτέλεση των δύο αυτών νημάτων το αρχικό νήμα-γονέας θα επιτελέσει την απαιτούμενη επεξεργασία (*process(node)*) στον δικό του κόμβο, μετά θα επιστρέψει τον έλεγχο στο πιο πάνω επίπεδο κοκ. Η παράμετρος *node* πρέπει να δείχνει αρχικά στη ρίζα του δέντρου, ενώ για την αρτιότερη εκτέλεση του αλγορίθμου ο αριθμός των νημάτων που δημιουργούνται για κάθε νέα ομάδα θα πρέπει να οριστεί ίσος με δύο (όσα είναι και τα *sections* που ανατίθενται σε κάθε αναδρομική κλήση).

6.2.6.4 Άλλα χρήσιμα παραδείγματα

Α. Υπολογισμός του Παραγοντικού (n!)

Ως ένα επιπλέον παράδειγμα για την καλύτερη κατανόηση του μηχανισμού του *reduction* που παρέχει το OpenMP, παρουσιάζεται παρακάτω η χρήση του για τον παράλληλο υπολογισμό του παραγοντικού ενός θετικού ακεραίου αριθμού.

`n! = 1 * 2 * 3 * … * (n-1) * n`

Για την παραλληλοποίηση του ανωτέρω υπολογισμού, αν θεωρήσουμε ότι έχουμε στη διάθεσή μας *t* νήματα, είναι εύκολο να παρατηρήσουμε ότι απαιτείται (α) ο

υπολογισμός ενός μέρους του συνόλου των πολλαπλασιασμών (*n/t* αν το *n* είναι ακέραιο πολλαπλάσιο του *t*) από κάθε νήμα, και (β) ο πολλαπλασιασμός των τελικών επιμέρους γινομένων που θα υπολογίσει το κάθε νήμα, όλων μεταξύ τους σε μία κοινή/σφαιρική μεταβλητή. Για την υλοποίηση δε του παραπάνω βήματος (β) μπορεί να χρησιμοποιηθεί ο μηχανισμός *reduction* με τον τελεστή του πολλαπλασιασμού '*'.

Ακολουθεί η υλοποίηση των παραπάνω σε ένα ολοκληρωμένο πρόγραμμα OpenMP (παρόμοιας μορφής με αυτό του υπολογισμού του εσωτερικού γινομένου δύο διανυσμάτων που είδαμε στην παράγραφο 6.2.6.1) με κατάλληλη χρήση της διαταγής διαμοιρασμού *for* και της δήλωσης *reduction*.

```
#include <omp.h>

main ()
{

int i, n, p, np;
int tid, result;
result = 1;

printf("Give an integer: ");
scanf("%d", &n);

#pragma omp parallel shared(n) private(i)
  {
   #pragma omp for schedule(static) reduction(*:result)
     for (i=1; i <= n; i++)
       result*=i;
  }

printf("Final Result: %d\n", result);
}
```

Αξίζει επίσης να σημειωθεί ότι ακόμα και για την περίπτωση που το *n* δεν είναι ακέραιο πολλαπλάσιο του *t*, ο προγραμματιστής δεν χρειάζεται να λάβει κάποια ειδικότερη πρόνοια. Το λογισμικό θα φροντίσει για την αρτιότερη δυνατή διαμοίραση, δίνοντας (αναπόφευκτα) σε κάποιο/α από τα νήματα μία λιγότερη επανάληψη. Επίσης σημειώστε ότι δεν χρειάζεται να ορίσουμε κάποια ειδικότερη τιμή για το μέγεθος (*chunk*) των ομάδων επαναλήψεων κατά τη διαμοίρασή τους στα νήματα. Υπονοείται εξ' ορισμού *chunk=1*, τιμή η οποία εξασφαλίζει και την αρτιότερη δυνατή διαμοίραση στην περίπτωση που το *n* δεν είναι ακέραιο πολλαπλάσιο του *t*).

Β. Συγχρονισμός/Επικοινωνία Νημάτων (PING-PONG)

Όπως αναφέρθηκε και στην παράγραφο 6.2.4 δεν παρέχεται από το OpenMP κάποιος ειδικότερος μηχανισμός για την υλοποίηση πιο σύνθετων αναγκών συγχρονισμού/επικοινωνίας μεταξύ των πολλαπλών νημάτων, όπως π.χ. απλής αναμονής και υπό συνθήκη αναμονής (condition synchronization). Για την υποστήριξη τέτοιου είδους αναγκών ωστόσο είναι δυνατή η χρήση των γενικότερων συναρτήσεων κλειδώματος (*locks*) τις οποίες παρέχει.

Σχετικά, παρουσιάζουμε παρακάτω ένα παράδειγμα απλής αναμονής με χρήση των ανωτέρω συναρτήσεων (κατά παρόμοιο τρόπο μπορούν να χρησιμοποιηθούν οι σχετικές συναρτήσεις και σε πιο σύνθετες περιπτώσεις αναμονής υπό συνθήκη). Έστω ότι μας ζητείται να συγχρονίσουμε δύο νήματα έτσι ώστε να τυπώνουν επαναληπτικά μία ακολουθία από εναλλασσόμενες εμφανίσεις των λεκτικών 'PING' και 'PONG', δηλαδή να παράγεται μία έξοδος της ακόλουθης μορφής:

```
PING  PONG  PING  PONG  PING ...
```

Για να το πετύχουμε αυτό δεν έχουμε παρά να σκεφτούμε με παρόμοιο τρόπο όπως αν χρησιμοποιούσαμε εργαλεία υλοποίησης σημαφόρων (π.χ. τα primitives *sem_post()* και *sem_wait()* των *posix semaphores*). Πιο συγκεκριμένα το παραπάνω πρόβλημα συντίθεται από δύο επιμέρους λειτουργίες αναμονής:

- Το $2^{ο}$ νήμα θα πρέπει να ξεκινάει κάθε επανάληψή του (για κάθε γύρο εκτύπωσης PING PONG) αφού ολοκληρωθεί η αντίστοιχη (του ιδίου γύρου εκτύπωσης) επανάληψη του $1^{ου}$ νήματος. Αυτό μπορεί να επιτευχθεί με χρήση ενός κλειδώματος που το $2^{ο}$ νήμα να το βλέπει δεσμευμένο μέχρι να τελειώσει την επανάληψή του το $1^{ο}$ νήμα (οπότε και θα το ελευθερώνει).
- Το $1^{ο}$ νήμα θα πρέπει να ξεκινάει κάθε επόμενη επανάληψή του (για τον επόμενο γύρο εκτύπωσης PING PONG) αφού ολοκληρωθεί η προηγούμενη επανάληψη του $2^{ου}$ νήματος. Αυτό μπορεί να επιτευχθεί με χρήση ενός ακόμα κλειδώματος που το $1^{ο}$ νήμα να το βλέπει δεσμευμένο μέχρι να τελειώσει την προηγούμενη επανάληψή του το $2^{ο}$ νήμα (οπότε και θα το ελευθερώνει).

Ένα ολοκληρωμένο πρόγραμμα σε OpenMP το οποίο πραγματοποιεί τα παραπάνω, με κατάλληλη χρήση της διαταγής *sections* για τη δημιουργία και ταυτόχρονη/παράλληλη εκτέλεση των δύο νημάτων, παρατίθεται παρακάτω:

```
#include <omp.h>

main ()
{
```

```
omp_lock_t lock1;
omp_lock_t lock2;
int i;

omp_init_lock(&lock1);
omp_unset_lock(&lock1);

omp_init_lock(&lock2);
omp_set_lock(&lock2);

#pragma omp parallel private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
     for (i=1; i<=20; i++)
      {
       omp_set_lock(&lock1);
       printf("PING\n");
       omp_unset_lock(&lock2);
      }
    #pragma omp section
     for (i=1; i<=20; i++)
      {
       omp_set_lock(&lock2);
       printf("PONG\n");
       omp_unset_lock(&lock1);
      }
    }
  }
}
```

Η αρχική τιμή του κλειδώματος του 1ου νήματος (*lock1*) θα πρέπει να τεθεί σε 'ελεύθερο' (μέσω της συνάρτησης *omp_unset_lock()*) ώστε κατά την πρώτη επανάληψη το 1ο νήμα να περάσει και να εκτελεστεί κανονικά. Η αρχική τιμή του κλειδώματος του 2ου νήματος (*lock2*) θα πρέπει να τεθεί σε 'δεσμευμένο' (μέσω της συνάρτησης *omp_set_lock()*), προκειμένου για να κάνει την πρώτη επανάληψή του το 2ο νήμα να πρέπει να περιμένει να τεθεί (ελευθερωθεί) το κλείδωμα από το 1ο νήμα. Θα πρέπει να σημειωθεί επίσης εδώ ότι σε ένα πρόγραμμα OpenMP μπορεί κάποιος αν το επιθυμεί να χρησιμοποιήσει χωρίς κανέναν περιορισμό (είτε αυτόνομα είτε σε συνδυασμό με τις συναρτήσεις κλειδώματος του OpenMP) για το συγχρονισμό/ επικοινωνία μεταξύ των νημάτων και *posix semaphores*.

Γ. Ταξινόμηση με τον αλγόριθμο Multisort

Όπως αναφέρθηκε και στην παράγραφο 6.2.5, μία από τις σημαντικότερες επιπλέον δυνατότητες τις οποίες προσέφερε ο νέος μηχανισμός των διακριτών εργασιών (*tasks*) στο OpenMP, είναι η πιο εύκολη διατύπωση και αποδοτική εκτέλεση παράλληλων αλγόριθμων που εκφράζονται σε αναδρομική μορφή (recursive procedures). Ως ένα ακόμα (λίγο πιο σύνθετο) παράδειγμα δίνεται παρακάτω μία ενδεικτική παράλληλη υλοποίηση με χρήση διακριτών εργασιών του αναδρομικού αλγόριθμου ταξινόμησης *multisort* ([FLR98]), ο οποίος αποτελεί μία παραλλαγή του γνωστού σε όλους μας αλγόριθμου ταξινόμησης με συγχώνευση (*mergesort*), και έχει αποδειχθεί πειραματικά ότι είναι ιδιαίτερα αποδοτικός όταν εκτελείται σε πολυπύρηνα (*multicore*) υπολογιστικά συστήματα.

Πιο συγκεκριμένα, ο αλγόριθμος *multisort* διαχωρίζει αρχικά την προς ταξινόμηση ακολουθία σε τέσσερα ισομεγέθη τμήματα, και συνεχίζει εφαρμόζοντας αναδρομικά την παραπάνω διαδικασία διαχωρισμού σε κάθε τμήμα. Οι αναδρομικές κλήσεις σταματούν όταν το μέγεθος του εξεταζόμενου τμήματος γίνεται αρκετά μικρό (μικρότερο από κάποιο συγκεκριμένο όριο το οποίο καθορίζεται από το χρήστη), οπότε σε εκείνο το σημείο απλά το εν λόγω τμήμα ταξινομείται με κάποιον γνωστό ακολουθιακό αλγόριθμο (π.χ. με τον αλγόριοθμο *quicksort*) και επιστρέφεται ο έλεγχος στο προηγούμενο επίπεδο κλήσης. Με την ολοκλήρωση κάθε αναδρομικής κλήσης, τα τέσσερα επιμέρους τμήματα τα οποία επιστρέφονται στο κυρίως σώμα ταξινομημένα, συγχωνεύονται σε μία ενιαία ταξινομημένη ακολουθία σε δύο βήματα (πρώτα συγχωνεύονται – παράλληλα – ανά δύο σε δύο τμήματα διπλάσιου μεγέθους, και στη συνέχεια τα δύο εναπομείναντα τμήματα συγχωνεύονται μεταξύ τους).

Η διατύπωση του ανωτέρω αλγορίθμου μπορεί να πραγματοποιηθεί πολύ κομψά, ευέλικτα και αποτελεσματικά [AC09], και τηρώντας πιστά τη φιλοσοφία η οποία υπαγορεύεται από την πρωτογενή αναδρομική αναπαράστασή του αλγορίθμου για ακολουθιακό περιβάλλον εκτέλεσης, με κατάλληλη χρήση των διαταγών *task* και *taskwait* που είδαμε στην παράγραφο 6.2.5, ως ακολούθως (η παράμετρος *start* αποτελεί δείκτη στην αρχή της προς ταξινόμηση ακολουθίας, η παράμετρος *space* αποτελεί δείκτη στην αρχή του χώρου μνήμης όπου θα αποθηκευτεί το τελικό αποτέλεσμα – όπου θα γίνουν δηλαδή σταδιακά οι υποδηλούμενες συγχωνεύσεις, και η παράμετρος *size* αναπαριστά το μέγεθος της ακολουθίας):

```
void multisort (start, space, size)
{
 if (size < limit)
   {
      quicksort (start, start+size-1);
```

```
      return();
   }

 quarter = size/4;

 startA = start;   spaceA = space;
 startB = startA + quarter;   spaceB = spaceA + quarter;
 startC = startB + quarter;   spaceC = spaceB + quarter;
 startD = startC + quarter;   spaceD = spaceC + quarter;

 #pragma omp task
    multisort (startA, spaceA, quarter);
 #pragma omp task
    multisort (startB, spaceB, quarter);
 #pragma omp task
    multisort (startC, spaceC, quarter);
 #pragma omp task
    multisort (startD, spaceD, size-3*quarter);

 #pragma omp taskwait

 #pragma omp task
    merge (startA, startA+quarter-1,
           startB, startB+quarter-1, spaceA);
 #pragma omp task
    merge (startC, startC+quarter-1,
           startD, start+size-1, spaceC);

 #pragma omp taskwait

 merge (spaceA, spaceC-1, spaceC, spaceA+size-1, startA);
}
```

Η χρήση/ορισμός διακριτών εργασιών διασφαλίζει το μέγιστο δυνατό βαθμό παραλληλισμού που υπαγορεύει ο αλγόριθμος (τόσο η ταξινόμηση των τεσσάρων επιμέρους τμημάτων κάθε αναδρομικής κλήσης όσο και το πρώτο βήμα της τελικής συγχώνευσης γίνονται με απόλυτα παράλληλο τρόπο). Προσέξτε ιδιαίτερα επίσης την απαραίτητη χρήση των διαταγών *taskwait* προκειμένου να τηρείται ο απαιτούμενος συγχρονισμός μεταξύ των πολλαπλών νημάτων/tasks.

Δ. Επίλυση της εξίσωσης Poisson με τη μέθοδο Jacobi

Όπως είδαμε στο κεφάλαιο 5 (παράγραφο 5.4.2) η μέθοδος *Jacobi* αποτελεί μία από τις ευρέως χρησιμοποιούμενες επαναληπτικές μεθόδους για την επίλυση γραμμικών συστημάτων (ειδικότερα με αραιούς πίνακες συντελεστών) καθώς επίσης και διαφορικών εξισώσεων (όπως π.χ. η εξίσωση *Poisson*). Από τη φύση της δε αποτελεί μία μέθοδο η οποία παραλληλοποιείται ιδιαίτερα εύκολα τόσο σε περιβάλλον

κοινής όσο και σε περιβάλλον κατανεμημένης μνήμης, καθώς μπορεί να γίνει εύκολα η αποδόμηση του συνολικού υπολογισμού σε ανεξάρτητες μεταξύ τους εργασίες για κάθε επανάληψη. Παρακάτω παραθέτουμε ένα ενδεικτικό απόσπασμα παράλληλης υλοποίησης σε OpenMP για την περίπτωση επίλυσης της εξίσωσης *Poisson,* στη μορφή που αυτή παρουσιάστηκε στα πλαίσια της παραγράφου 5.4.2.

```
void poisson_estimation
(int n, double h, double f[n][n], int start, int end,
double u[n][n], double unew[n][n])

{
  int i,j;
  int index;

# pragma omp parallel shared(h, f, start, end, n, u, unew)
                      private(i, j, index)

for (index = start+1; index <= end; index++)
{

# pragma omp for schedule(static)
    for (j=0; j<n; j++)
    {
      for (i=0; i<n; i++)
      {
        u[i][j] = unew[i][j];
      }
    }

// # pragma omp barrier (?)

# pragma omp for schedule(static)
    for (j=0; j<n; j++)
    {
      for (i=0; i<n; i++)
      {
        if (i==0 || j==0 || i==n-1 || j==n-1)
        {
          unew[i][j] = f[i][j];
        }
        else
        {
          unew[i][j] = 0.25 * (
             u[i-1][j] + u[i][j+1] + u[i][j-1] + u[i+1][j]
             + f[i][j] * h * h );
        }
      }
    }
  }
}
```

Σχετικά με την ανωτέρω παράλληλη υλοποίηση έχουμε υποθέσει (πράγμα που μπορεί να εφαρμοστεί χωρίς να προκαλεί άξια λόγου επιβάρυνση στον τελικό αριθμό επαναλήψεων σε μεγάλης κλίμακας συστήματα) ότι μεσολαβεί σημαντικός αριθμός επαναλήψεων (λ=*end-start*) πριν επανελεγχθεί αν συγκλίνει η μέχρι εκείνη τη στιγμή υπολογισθείσα λύση του συστήματος ή όχι. Μπορούμε να θεωρήσουμε δηλαδή ότι η κλήση της συνάρτησης που παρουσιάστηκε παραπάνω (*poisson_estimation()*) γίνεται επαναληπτικά από το κυρίως σώμα ενός προγράμματος το οποίο καθορίζει τι εμφανιζόμενες παραμέτρους (μεταξύ αυτών και τα *end* και *start* καθώς και τις αρχικές τιμές για την άρτια εκκίνηση των υπολογισμών κοκ) και επιστρέφει σε αυτό ο έλεγχος μετά από κάθε λ επαναλήψεις προκειμένου να ελεγχθεί η σύγκλιση. Η επιλογή αυτή προάγει μεταξύ άλλων και την απόδοση της παράλληλης εκτέλεσης καθώς θα καταργούνται και επανεκκινούνται νέες ομάδες νημάτων (κάθε φορά που καλείται η συνάρτηση) σε πολύ πιο αραιά χρονικά διαστήματα. Έχουμε υποθέσει επίσης τα ακόλουθα για τις χρησιμοποιούμενες παραμέτρους:

- *n*: η διάσταση του πλέγματος (θεωρούμε *nxn grid*).
- *h*: η απόσταση των σημείων του πλέγματος μεταξύ τους.
- *start* και *end*: οι δείκτες (αύξοντες αριθμοί) της πρώτης και της τελευταίας επανάληψης για την τρέχουσα κλήση της συνάρτησης.
- *f[n][n]*: ο πίνακας με τα δεδομένα (τιμές) της δεξιάς πλευράς του συστήματος.
- *u[n][n]*: ο πίνακας με τις τρέχουσες τιμές (υπολογισθείσες προσεγγιστικές λύσεις) του συστήματος (αυτές της προηγούμενης κάθε φορά επανάληψης).
- *unew[n][n]*: ο πίνακας με τις νέες τιμές (προσεγγιστικές λύσεις) του συστήματος (αυτές που θα υπολογιστούν στην τρέχουσα επανάληψη).

Στο παραπάνω πλαίσιο, εκκινείται αρχικά μία ομάδα νημάτων μέσω της διαταγής *parallel*, κάθε ένα από τα οποία εκτελεί το *for-loop* που ακολουθεί στην αμέσως επόμενη γραμμή. Σε κάθε επανάληψη αυτού του *for-loop,* όταν συναντάται μία παράλληλη διαταγή *for*, κάθε ένα από τα ήδη δημιουργηθέντα νήματα λαμβάνει ένα αριθμό από τις υποδηλούμενες εσωτερικές επαναλήψεις (με στατική μία-προς-μία διαμοίραση αυτών) και συνεχίζει με αυτές την εκτέλεσή του.

Αξίζει να σημειωθεί επίσης εδώ ότι μεταξύ των δύο παράλληλων υπολογισμών οι οποίοι επιτελούνται από τις δύο παράλληλες διαταγές *for*, είναι απαραίτητο να υπάρχει/διασφαλίζεται συγχρονισμός τύπου φράγματος (*barrier*). Πριν δηλαδή αρχίσει από οποιοδήποτε νήμα ο υπολογισμός των νέων τιμών στον πίνακα *unew* θα πρέπει να έχουν μεταφερθεί οι προηγούμενες τιμές του πλήρως στον πίνακα *u*

(ώστε και να μη χαθεί/επανεγγραφεί κάποια από αυτές και να λειτουργήσουν και ως οι παλαιές τιμές για τη νέα επανάληψη). Το παραπάνω επιτυγχάνεται στον παρατιθέμενο κώδικα μέσω της χρήσης δύο ξεχωριστών διαταγών *for* για την επιτέλεση των δύο αυτών ξεχωριστών υπολογισμών, καθώς όπως έχει τονισθεί και σε προηγούμενη παράγραφο (παρ. 6.2.3) στο τέλος κάθε παράλληλης διαταγής διαμοιρασμού εργασίας (όπως είναι και η *for*) υλοποιείται εξ' ορισμού από το ίδιο το OpenMP συγχρονισμός τύπου φράγματος για όλα τα εκτελούμενα νήματα. Έτσι δεν χρειάζεται να εισάγουμε οι ίδιοι κάποια διαταγή *barrier* (όπως έχει σημειωθεί ενδεικτικά μέσα σε σχόλια στον παραπάνω κώδικα) ανάμεσα στις δύο αυτές διαταγές *for*.

Αντίστοιχος συγχρονισμός θα πρέπει επίσης εμφανώς να διασφαλίζεται και πριν κάθε νήμα ξεκινήσει την κάθε επόμενη επανάληψή του (θα πρέπει δηλαδή όλα τα νήματα να έχουν ολοκληρώσει την προηγούμενη επανάληψή τους), έτσι ώστε να είναι σίγουρο ότι θα ενημερωθεί ο πίνακας *u* με τις συνεπείς νέες τιμές της προηγούμενης επανάληψης (και δεν θα χαθεί κάποια από αυτές). Είναι εύκολο να διαπιστωθεί με βάση αναφερόμενα παραπάνω ότι και αυτή η απαίτηση συγχρονισμού ικανοποιείται από τον παρατιθέμενο κώδικα, μέσω της εξ' ορισμού υλοποίησης συγχρονισμού φράγματος στο τέλος της δεύτερης διαταγής *for*.

Για περισσότερες λεπτομέρειες και για την πλήρη αντίστοιχη υλοποίηση μπορεί κανείς να ανατρέξει στο [Qu03]. Με αντίστοιχο τρόπο επίσης είναι εύκολο να πραγματοποιηθεί (αφήνεται σαν άσκηση) και η παραλληλοποίηση σε OpenMP της μεθόδου *Jacobi* για την επίλυση γραμμικών συστημάτων (βλ. παρ. 5.4.2).

6.2.6.5 Άλλες συναρτήσεις βιβλιοθήκης του OpenMP

Πέραν των βασικών συναρτήσεων βιβλιοθήκης για τον αριθμό και τα *ids* των νημάτων (*omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num()*) και τις συναρτήσεις διαχείρισης κλειδωμάτων (*locks*), τις οποίες έχουμε ήδη παρουσιάσει στις προηγούμενες παραγράφους, το OpenMP προσφέρει και έναν ακόμα σημαντικό αριθμό συναρτήσεων βιβλιοθήκης σε χρόνο εκτέλεσης, οι βασικότερες από τις οποίες παρουσιάζονται συνοπτικά παρακάτω:

```
int omp_get_max_threads(void);
```

Επιστρέφει τον μέγιστο αριθμό νημάτων που μπορούν να δοθούν σε μία νέα ομάδα νημάτων ως αποτέλεσμα μιας νέας κλήσης της διαταγής *parallel*.

```
int omp_get_thread_limit(void)
```

Επιστρέφει τον μέγιστο συνολικά (σε αντιδιαστολή με την προαναφερθείσα) αριθμό νημάτων που είναι διαθέσιμα εξαρχής στο πρόγραμμα.

`int omp_get_num_procs(void);`
Επιστρέφει τον αριθμό των διαθέσιμων επεξεργαστών/πυρήνων του συστήματος.

`int omp_in_parallel(void);`
Επιστρέφει *αληθές* αν η κλήση έχει γίνει εντός μίας ενεργής παράλληλης περιοχής, ειδάλλως επιστρέφει *ψευδές*.

`void omp_set_dynamic(int dyn_threads);`
Ενεργοποιεί (*dyn_threads*≠0) ή απενεργοποιεί (*dyn_threads*=0) τη δυνατότητα δυναμικού προσδιορισμού του αριθμού των διαθέσιμων νημάτων, θέτοντας ανάλογα την εσωτερική μεταβλητή ελέγχου (ICV – internal control variable) *dyn-var.*

`int omp_get_dynamic(void);`
Επιστρέφει την τιμή της εσωτερικής μεταβλητής ελέγχου (ICV) *dyn-var*, η οποία προσδιορίζει αν η δυνατότητα δυναμικού προσδιορισμού του αριθμού των διαθέσιμων νημάτων είναι ενεργοποιημένη ή απενεργοποιημένη.

`void omp_set_nested(int nested);`
Ενεργοποιεί (*nested*≠0) ή απενεργοποιεί (*nested*=0) τη δυνατότητα εμφωλιασμένου παραλληλισμού, θέτοντας ανάλογα την εσωτερική μεταβλητή ελέγχου (ICV) *nest-var*.

`int omp_get_nested(void);`
Επιστρέφει την τιμή της εσωτερικής μεταβλητής ελέγχου (ICV) *nest-var*, η οποία προσδιορίζει αν η δυνατότητα εμφωλιασμένου παραλληλισμού είναι ενεργοποιημένη ή απενεργοποιημένη.

`void omp_set_max_active_levels(int max_levels);`
Περιορίζει τον μέγιστο επιτρεπόμενο αριθμό επιπέδων ενεργών εμφωλιασμένων παράλληλων περιοχών (parallel regions), θέτοντας ανάλογα την εσωτερική μεταβλητή ελέγχου (ICV) *max-active-levels-var*.

`int omp_get_max_active_levels(void);`
Επιστρέφει την τιμή της εσωτερικής μεταβλητής ελέγχου (ICV) *max-active-levels-var*, η οποία προσδιορίζει τον μέγιστο επιτρεπόμενο αριθμό επιπέδων ενεργών εμφωλιασμένων παράλληλων περιοχών.

`int omp_get_level(void);`
Επιστρέφει τον αριθμό των εμφωλιασμένων παράλληλων περιοχών από τις οποίες περικλείεται η διακριτή εργασία η οποία έκανε την κλήση.

```
int omp_get_active_level(void);
```
Επιστρέφει τον αριθμό των εμφωλιασμένων ενεργών παράλληλων περιοχών από τις οποίες περικλείεται η διακριτή εργασία η οποία έκανε την κλήση.

```
int omp_get_ancestor_thread_num(int level);
```
Επιστρέφει τον αριθμό του νήματος-προγόνου του τρέχοντος νήματος, σε επίπεδο εμφωλιασμού αυτό που προσδιορίζεται από την παράμετρο *level*.

```
int omp_get_team_size(int level);
```
Επιστρέφει το πλήθος των νημάτων της ομάδας του νήματος-προγόνου του τρέχοντος νήματος, σε επίπεδο εμφωλιασμού όπως προσδιορίζεται από την παράμετρο *level*.

```
double omp_get_wtime(void);
```
Επιστρέφει μία χρονική ένδειξη (πόσος χρόνος έχει περάσει) σε δευτερόλεπτα από κάποιο πρότερο σταθερό χρονικό σημείο.

```
double omp_get_wtick(void);
```
Επιστρέφει την ακρίβεια (πόσος χρόνος μεσολαβεί μεταξύ δύο ticks) του μετρητή/ ρολογιού που χρησιμοποιείται από τη συνάρτηση *omp_get_wtime()*.

6.2.6.6 Μετάφραση προγραμμάτων OpenMP σε C/C++

Με χρήση του μεταφραστή *gcc* μπορεί κανείς πολύ εύκολα να μεταφράσει ένα πρόγραμμα OpenMP σε C/C++, όπως θα το έκανε για ένα απλό πρόγραμμα C/C++ και θέτοντας επιπλέον τη σημαία *–fopenmp*. Δηλαδή π.χ. για τη μετάφραση ενός αρχείου *omp_test.c* αρκεί η ακόλουθη εντολή:

```
gcc –o omp_test omp_test.c -fopenmp
```

Η τελευταία υποστηριζόμενη έκδοση από τον μεταφραστή *gcc* (από την έκδοση 4.7 και μετά αυτού) είναι η έκδοση OpenMP 3.1. Από την έκδοση *gcc* 4.4 και μετά υποστηρίζεται η έκδοση OpenMP 3.0, ενώ από την έκδοση *gcc* 4.2 και μετά υποστηρίζεται η έκδοση OpenMP 2.5. Πρόσφατα (Ιούλιος 2013) έχει ανακοινωθεί επίσης η έκδοση OpenMP 4.0 (με αρκετά νέα χαρακτηριστικά), η οποία δεν έχει ακόμα ενσωματωθεί στον *gcc* (αναμένεται μέσα στο επόμενο έτος). Σε περιβάλλον Windows (Microsoft Visual Studio 2008/2010) αντίστοιχα, υποστηρίζεται μέχρι σήμερα η έκδοση OpenMP 2.0. Για περισσότερες λεπτομέρειες και πιο επίκαιρη ενημέρωση σχετικά με την ενσωμάτωση νεώτερων εκδόσεων στους παραπάνω μεταφραστές (αλλά και σε άλλους υπάρχοντες) μπορεί κανείς να ανατρέξει στην επίσημη ιστοσελίδα του OpenMP (http://openmp.org).

ΒΙΒΛΙΟΓΡΑΦΙΑ

[IPP01] Introduction to Parallel Programming Tutorial, Lawrence Livermore National Laboratories, https://computing.llnl.gov/tutorials/parallel_comp/

[OMP01] OpenMP API Specification, http://openmp.org/wp/

[OMP02] OpenMP Tutorial, Lawrence Livermore National Laboratories, https://computing.llnl.gov/tutorials/openMP/

[MPI01] MPI Tutorial, Lawrence Livermore National Laboratories, https://computing.llnl.gov/tutorials/mpi/

[MPI02] MPI Man Pages, http://www.mcs.anl.gov/research/projects/mpi/www/

[MPI03] MPI Forum, http://www.mpi-forum.org/

[Qu03] M. Quinn, *Parallel Programming in C with MPI and OpenMP*, Mc Graw Hill, 2003.

[Pa96] P. Pacheco, Parallel Programming with MPI, Morgan-Kaufmann, 1996.

[FLR98] M. Frigo, C. Leiserson, K. Randall, The implementation of the Cilk-5 multithreaded language. In Proc. of ACM SIGPLAN PLDI '98 Conference, pp. 212–223, New York, USA, 1998.

[AC09] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, G. Zhang,The Design of OpenMP Tasks, IEEE TPDS, Vol. 20, No. 3, 2009