Πανεπιστημιο Πειραιως

Σχολη Τεχνολογιων Πληροφορικης και Επικοινωνιων

Τμημα Πληροφορικης

Σχεδιαση και Αναλυση Αλγοριθμων

Χαραλαμπος Κωνσταντοπουλος

ΠΜΣ Προηγμενα Συστηματα Πληροφορικης

Οκτωβριος 2014

Περιεχόμενα

1 Εισαγωγή **9**
- 1.1 Ανάγκη για υπολογιστική δύναμη . . . 9
- 1.2 Παράλληλος υπολογισμός . . . 10
  - 1.2.1 Παράγοντας επιτάχυνσης . . . 10
  - 1.2.2 Μέγιστη επιτάχυνση - Νόμος του Amdahl . . . 10
  - 1.2.3 Παράδειγμα Υπεργραμμικής Επιτάχυνσης - Αλγόριθμοι Αναζήτησης . . . 11
- 1.3 Παράλληλος Προγραμματισμός . . . 12
  - 1.3.1 Πολυεπεξεργαστές (Multiprocessors) . . . 14

2 Προγραμματισμός με πέρασμα μηνυμάτων **17**
- 2.1 Σύγχρονο (Synchronous) πέρασμα μηνυμάτων . . . 18
- 2.2 Ασύγχρονο (asynchronous) πέρασμα μηνυμάτων . . . 19
  - 2.2.1 Ετικέτα μηνύματος . . . 20
- 2.3 Διασπορά - Scatter . . . 20
- 2.4 Συγκέντρωση - Gather . . . 21
- 2.5 Μείωση - Reduce . . . 21
- 2.6 MPI (Message Passing Interface) . . . 22
  - 2.6.1 Communicators . . . 22
  - 2.6.2 Πιθανά σενάρια εκτέλεσης των συναρτήσεων send και receive . . . 23
  - 2.6.3 MPI - Επικοινωνία Σημείο σε Σημείο (Point-to-Point) . . . 23
  - 2.6.4 Nonblocking MPI συναρτήσεις . . . 24
  - 2.6.5 Συλλεκτική (Collective) Επικοινωνία . . . 25
  - 2.6.6 Barrier (Φράγμα) . . . 26
- 2.7 Αποτίμηση απόδοσης παράλληλων προγραμμάτων . . . 27
  - 2.7.1 Χρόνος Υπολογισμού . . . 27
  - 2.7.2 Χρόνος Επικοινωνίας . . . 27
  - 2.7.3 Δείκτες Απόδοσης . . . 28

3 Πολύ εύκολες παραλληλοποιήσεις **29**
- 3.1 Επεξεργασία εικόνας χαμηλού επιπέδου . . . 30
- 3.2 Το σύνολο Mandelbrot . . . 31
  - 3.2.1 Παράλληλη υλοποίηση του υπολογισμού του συνόλου Mandelbrot . . . 31
- 3.3 Μέθοδοι Monte Carlo . . . 33
  - 3.3.1 Υπολογισμός Ολοκληρώματος με τη μέθοδο Monte Carlo . . . 34

4 Στρατηγικές Διαμέρισης (Partitioning) και Διαίρει και Βασίλευε (Divide-and-Conquer) **37**
- 4.1 Παράδειγμα διαμέρισης ; Άθροισμα των στοιχείων μίας λίστας . . . 37
- 4.2 Τεχνική Διαίρει και Βασίλευε . . . 38
- 4.3 Bucketsort . . . 39

4.3.1 Παράλληλη υλοποίηση του bucket sort - Απλή προσέγγιση 40
4.3.2 Παράλληλη υλοποίηση του bucket sort - Καλύτερη παράλληλη λύση 40
4.4 Αριθμητική ολοκλήρωση με τη χρήση ορθογωνίων 41
4.5 Αριθμητική ολοκλήρωση με τη χρήση τραπεζίων 41
4.6 Προσαρμοζόμενος Τετραγωνισμός (Adaptive Quadrature) 42
4.7 Υπολογισμός της τιμής του π 43
4.8 Το πρόβλημα N-Body 45
4.8.1 Ακολουθιακός κώδικας για το πρόβλημα N-body 46
4.8.2 Παράλληλη υλοποίηση για το πρόβλημα N-body 46

5 Υπολογισμοί με την τεχνική της σωλήνωσης 49
5.1 Υπολογιστικές πλατφόρμες κατάλληλες για την τεχνική της σωλήνωσης 52
5.2 Παραδείγματα λύσεων με βάση την τεχνική της σωλήνωσης 52
5.2.1 Άθροιση αριθμών - Σωλήνωση Τύπου 1 52
5.2.2 Ταξινόμηση - Σωλήνωση Τύπου 2 54
5.2.3 Παραγωγή πρώτων αριθμών - Τύπος 2 σωλήνωσης 55
5.2.4 Επίλυση συστήματος γραμμικών εξισώσεων (άνω τριγωνική μορφή) - Τύπος 3 σωλήνωσης 56

6 Συγχρονισμένοι υπολογισμοί 59
6.1 Υλοποίηση φράγματος 60
6.1.1 Υλοποίηση του γραμμικού φράγματος με master-slave τεχνική 61
6.1.2 Υλοποίηση φράγματος με δέντρο 62
6.1.3 Υλοποίηση φράγματος με πεταλούδα 62
6.2 Τοπικός Συγχρονισμός 63
6.3 Αδιέξοδο (deadlock) 63
6.4 Κατηγορίες συγχρονισμένου υπολογισμού 64
6.4.1 Πλήρως συγχρονισμένοι υπολογισμοί 64
6.4.2 Τοπικός Σύγχρονος Υπολογισμός 69
6.5 Κυψελιδικά Αυτόματα (Cellular Automata) 75
6.6 Μερικώς συγχρονισμένοι υπολογισμοί 76

7 Εξισορρόπηση φόρτου και ανίχνευση τερματισμού 79
7.1 Στατική εξισορρόπηση φόρτου 79
7.2 Δυναμική εξισορρόπηση φόρτου 80
7.2.1 Συγκεντρωτική δυναμική εξισορρόπηση φόρτου 81
7.2.2 Τερματισμός 81
7.2.3 Αποκεντρωμένη Δυναμική Εξισορρόπηση Φόρτου 81
7.2.4 Μέθοδος εκκίνησης από τη πλευρά του λήπτη 83
7.2.5 Μέθοδος εκκίνησης από τη πλευρά του αποστολέα 83
7.2.6 Εξισορρόπηση φορτίου με διεργασίες σε γραμμική διάταξη 83
7.3 Κατανεμημένη Ανίχνευση Τερματισμού 85
7.3.1 Αλγόριθμος Ανίχνευσης Τερματισμού με διεργασίες σε διάταξη δακτυλίου ενός περάσματος 87
7.3.2 Αλγόριθμος Τερματισμού δύο περασμάτων με τις διεργασίες σε διάταξη δακτυλίου 87
7.3.3 Αλγόριθμος τερματισμού με διεργασίες σε διάταξη δένδρου 88
7.3.4 Αλγόριθμος κατανεμημένου τερματισμού με σταθερή ενέργεια 88
7.3.5 Παράδειγμα εξισορρόπησης φόρτου και ανίχνευσης τερματισμού 89

8 Προγραμματισμός σε συστήματα διαμοιραζόμενης μνήμης **95**
8.1 Πολυεπεξεργαστικά συστήματα διαμοιραζόμενης μνήμης 95
8.2 Διαμοιραζόμενα δεδομένα σε συστήματα με ςαςηε μνήμες 96
8.2.1 Ψευδής Διαμοιρασμός (False Sharing) 97
8.3 Προγραμματισμός σε Συστήματα Διαμοιραζόμενης Μνήμης 98
8.3.1 Χρήση «βαριών» διεργασιών 98
8.3.2 Τεχνική fork/join 98
8.3.3 Διεργασίες και νήματα (threads) 99
8.3.4 Απομονωμένα (detached) νήματα 101
8.3.5 Ταυτοχρονισμός - Concurrency 102
8.3.6 Thread-Safe Routines (Ρουτίνες ασφαλείς σε περιβάλλον νημάτων) 103
8.3.7 Προσπέλαση Διαμοιραζόμενων Δεδομένων 103
8.3.8 Κρίσιμο Τμήμα (Critical Section) 103
8.3.9 Λειτουργίες Κλειδώματος του Προτύπου Pthread 104
8.3.10 Αδιέξοδο (Deadlock) 106
8.3.11 Σημαφόροι (Semaphores) 106
8.3.12 Μεταβλητές Συνθήκης (Condition Variables) 107
8.3.13 OpenMP 109
8.3.14 Οδηγίες συγχρονισμού 113
8.4 Παραδείγματα προγραμμάτων διαμοιραζόμενης μνήμης 114
8.4.1 Θέσεις μνήμης με πρόσβαση και από τις δύο διεργασίες UNIX 115

9 Αλγόριθμοι ταξινόμησης **121**
9.1 Χωρισμός Δεδομένων (Data Partitioning) 122
9.2 Αλγόριθμος Bubble Sort 123
9.2.1 Παράλληλος αλγόριθμος Bubble Sort 123
9.3 Αλγόριθμος ταξινόμησης Odd-Even (Transposition) Sort 124
9.4 Αλγόριθμος Mergesort 124
9.5 Ο αλγόριθμος Quicksort 126
9.5.1 Εναλλακτικός τρόπος υλοποίησης του Quicksort σε σύστημα διαμοιραζόμενης μνήμης 127
9.6 Ο Αλγόριθμος Odd-Even Mergesort 128
9.7 Ο Αλγόριθμος Bitonic Mergesort 129
9.8 Αλγόριθμοι Bucket sort και Sample sort 130

10 Αριθμητικοί Αλγόριθμοι **135**
10.1 Άθροιση και Πολλαπλασιασμός Πινάκων 135
10.2 Χωρισμός Πινάκων σε Υποπίνακες 136
10.2.1 Αναδρομική υλοποίηση του πολλαπλασιασμού πινάκων 138
10.2.2 Ο αλγόριθμος του Cannon για πολλαπλασιασμό πινάκων 139
10.3 Επίλυση συστήματος γραμμικών εξισώσεων 140
10.3.1 Επίλυση γραμμικού συστήματος με τη μέθοδο Gaussian Elimination 140
10.3.2 Επαναληπτικές μέθοδοι επίλυσης γραμμικών συστημάτων 143
10.4 Διάταξη Red-Black 144

11 Αλγόριθμοι Γραφημάτων **147**
11.1 Ορισμοί 147
11.2 Ο αλγόριθμος του Prim 148
11.2.1 Παράλληλη υλοποίηση του αλγόριθμου Prim 149
11.3 Εύρεση συντομότερων μονοπατιών με κοινή αφετηρία 150

11.3.1 Παράλληλη υλοποίηση του αλγόριθμου του Dijkstra 151
11.4 Εύρεση συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών 151
11.5 Αλγόριθμος του Floyd 152
11.5.1 Παράλληλη υλοποίηση του αλγόριθμου του Floyd 153
11.6 Μεταβατική κλειστότητα 155
11.7 Εύρεση συνεκτικών συνιστωσών: Αναζήτηση πρώτα κατά βάθος 155
11.8 Εύρεση ενός Μέγιστου (Maximal) Ανεξάρτητου Συνόλου 156
11.9 Συντομότερα μονοπάτια με κοινή αφετηρία 158
11.10 Το Πρόβλημα του Περιοδεύοντος Πωλητή (ΠΠΠ) 159
11.10.1 Υπολογισμός του κόστους του τρέχοντος καλύτερου γύρου πόλεων 160
11.10.2 Δυναμική παραλληλοποίηση της αναζήτησης δέντρου χρησιμοποιώντας Pthreads 160
11.11 Παράλληλη υλοποίηση της αναζήτησης δέντρου με OpenMP 161
11.12 Υλοποίηση της αναζήτησης δέντρου χρησιμοποιώντας MPI και στατική διαμέριση . 162

12 Βιβλιογραφία 169

Πρόλογος

Στο τρέχον εξάμηνο, βασικό αντικείμενο του μαθήματος είναι ο παράλληλος υπολογισμός, οι αλγοριθμικές τεχνικές που εκμεταλλεύονται τις νέες αρχιτεκτονικές υπολογιστών καθώς τα προγραμματιστικά περιβάλλοντα για την ανάπτυξη αυτών των εφαρμογών. Στόχος των διαλέξεων είναι ο φοιτητής να αποκτήσει το απαραίτητο υπόβαθρο ώστε στη συνέχεια να μπορεί να εξειδικεύσει περισσότερο στη περιοχή. Θα παρουσιαστούν τεχνικές και προγραμματιστικά περιβάλλοντα τόσο για κατανεμημένα σύστημα επεξεργασίας όσο και για πολυεπεξεργαστικά συστήματα όπως οι σύγχρονοι πολυπύρηνοι επεξεργαστές. Μεγάλο τμήμα των διαλέξεων έχει βασισθεί στο [1] και στις διαφάνειες που το συνοδεύουν. Επίσης σημαντικά τμήματα της ύλης προέρχονται και από τις υπόλοιπες αναφορές της βιβλιογραφίας που παρατίθεται στο τέλος των σημειώσεων.

Επίσης, θα ήθελα να ευχαριστήσω τον Κώστα Μανέ, διδάκτορα του Τμήματος και Εργαστηριακό Συνεργάτη του μαθήματος για τη συμβολή του στην προετοιμασία αυτών των σημειώσεων.

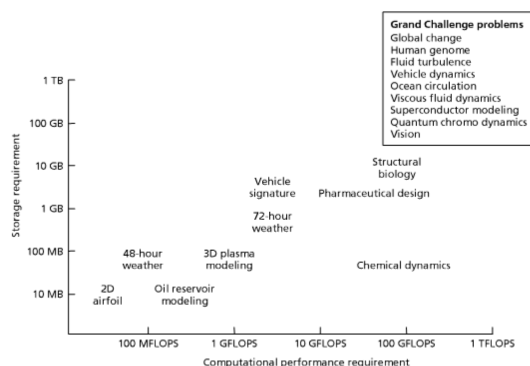
Πειραιάς, Οκτώβριος 2014

Κεφάλαιο 1

Εισαγωγή

1.1 Ανάγκη για υπολογιστική δύναμη

Η ανάγκη για όλο και μεγαλύτερη ταχύτητα στους υπολογισμούς δεν μπορεί να ικανοποιηθεί πλέον από μονοεπεξεργαστικά συστήματα. Οι περιοχές που απαιτούν υψηλότερες ταχύτητες υπολογισμού είναι μεταξύ άλλων οι περιοχές του επιστημονικού υπολογισμού, η προσομοίωση επιστημονικών προβλημάτων καθώς και προβλημάτων μηχανικής. Σε τέτοια προβλήματα, οι υπολογισμοί πρέπει να ολοκληρώνονται σε ένα εύλογο χρονικό διάστημα. Τα προβλήματα αυτά είναι επίσης γνωστά και ως “Grand Challenge problems”.



Σχήμα 1.1: Προβλήματα με μεγάλο όγκο επεξεργασίας.

Παράδειγμα πρόβλεψης καιρού. Υποθέτουμε ότι όλη η ατμόσφαιρα της γης χωρίζεται σε κυψέλες διαστάσεων $1Km{\times}1Km{\times}1Km$, μέχρι το ύψος των $10Km$ (10 κυψέλες σε ύψος). Συνολικά, θα έχουμε 5×10^8 κυψέλες. Υποθέτουμε κάθε υπολογισμός απαιτεί 200 λειτουργίες κινητής υποδιαστολής. Σε ένα βήμα υπολογισμού, 10^{11} τέτοιες λειτουργίες είναι απαραίτητες. Για να προβλέψουμε τον καιρό για μετά από 7 ημέρες χρησιμοποιώντας διαστήματα του ενός λεπτού, ένας υπολογιστής που εκτελεί 1Gflops (10^9 λειτουργίες κινητής υποδιαστολής/δευτερόλεπτο) χρειάζεται 10^6 δευτερόλεπτα ή πάνω από 10 ημέρες. Για να εκτελέσουμε ένα υπολογισμό σε 5 λεπτά απαιτούνται υπολογιστές που λειτουργούν στα 3.4 Tflops (3.4×10^{12} λειτουργίες κινητής υποδιαστολής/δευτερόλεπτο).

Μοντελοποίηση της κίνησης Ουρανίων Σωμάτων. Κάθε σώμα έλκεται από κάθε άλλο σώμα από τους νόμους του Νεύτωνα. Η ελκτική δύναμη μεταξύ δύο σωμάτων είναι αντιστρόφως

ανάλογη του τετραγώνου της απόστασής τους. Η κίνηση κάθε σώματος καθορίζεται από τη συνισταμένη των δυνάμεων που ασκούνται σε κάθε σώμα. Αν έχουμε N σώματα, πρέπει να υπολογίσουμε $N-1$ δυνάμεις για κάθε σώμα και συνολικά σχεδόν N^2 υπολογισμούς. Χρησιμοποιώντας ένα καλό προσεγγιστικό αλγόριθμο για το συγκεκριμένο πρόβλημα, μπορούμε να μειώσουμε την πολυπλοκότητα σε $N \log N$. Μετά τη μετακίνηση των σωμάτων στη νέα τους θέση τους, ο παραπάνω υπολογισμός πρέπει να επαναληφθεί, αφού οι ασκούμενες δυνάμεις μεταξύ των σωμάτων έχουν αλλάξει. Για παράδειγμα, ένας γαλαξίας μπορεί να έχει 10^{11} αστέρια. Ακόμα και αν υποθέσουμε ότι κάθε υπολογισμός χρειάζεται ένα $1ms$ για να ολοκληρωθεί, συνολικά χρειάζονται 10^9 χρόνια για κάθε επανάληψη χρησιμοποιώντας τον αλγόριθμο πολυπλοκότητας $O(N^2)$ ή σχεδόν ένα χρόνο για μία επανάληψη χρησιμοποιώντας τον αλγόριθμο πολυπλοκότητας $O(N \log N)$.

1.2 Παράλληλος υπολογισμός

Επίλυση υπολογιστικών προβλημάτων με τη χρήση περισσότερων από έναν υπολογιστή ή ενός υπολογιστής με περισσότερους από έναν επεξεργαστή. Το βασικό κίνητρο είναι συνήθως η ολοκλήρωση των υπολογισμών σε συντομότερο χρόνο. Ιδανικά, με n υπολογιστικά στοιχεία που δουλεύουν ταυτόχρονα, ο συνολικός χρόνος υπολογισμού μειώνεται n φορές. Στην πράξη, η επιτάχυνση στον υπολογισμό θα είναι μικρότερη από n. Άλλα κίνητρα για παράλληλο υπολογισμό είναι η ανοχή στα σφάλματα, περισσότερη διαθέσιμη μνήμη κτλ.

1.2.1 Παράγοντας επιτάχυνσης

Ο παράγοντας επιτάχυνσης ορίζεται ως

$$S(p) = \frac{\text{Χρόνος εκτέλεσης σε ένα επεξεργαστή}}{\text{Χρόνος εκτέλεσης σε ένα πολυεπεξεργαστή με } p \text{ επεξεργαστές}} = \frac{t_s}{t_p}$$

όπου t_s είναι ο χρόνος εκτέλεσης σε ένα επεξεργαστή και t_p είναι ο χρόνος εκτέλεσης σε ένα πολυεπεξεργαστή με p επεξεργαστές. Ο παράγοντας επιτάχυνσης $S(p)$ δίνει την αύξηση στη ταχύτητα υπολογισμού με τη χρήση επεξεργαστή. Για το μονό επεξεργαστή, χρησιμοποιούμε τον καλύτερο ακολουθιακό αλγόριθμο. Ο αλγόριθμος στη παράλληλη υλοποίηση μπορεί να είναι και συνήθως είναι διαφορετικός.

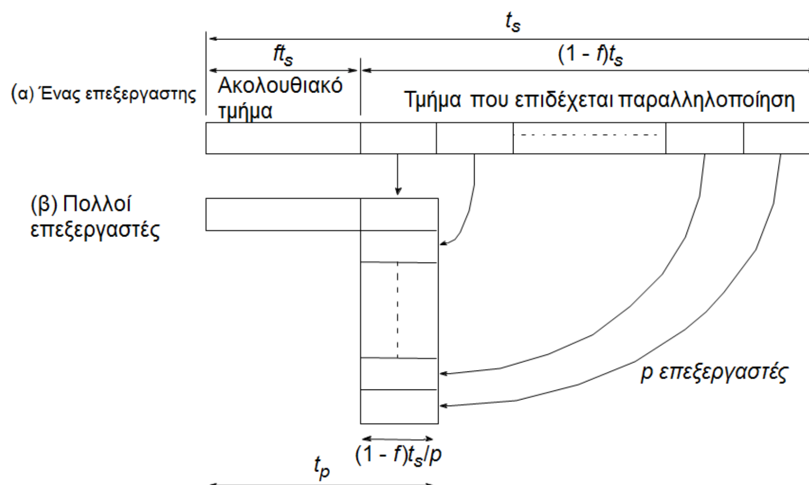
Η μέγιστη επιτάχυνση που μπορούμε να επιτύχουμε με p επεξεργαστές είναι συνήθως p (γραμμική επιτάχυνση). Είναι επίσης πιθανόν να λάβουμε υπερ-γραμμική (superlinear) επιτάχυνση (μεγαλύτερη από p). Αυτό συνήθως οφείλεται στην επιπλέον μνήμη του πολυεπεξεργαστικού συστήματος ή/και στην καλύτερη απόδοση των cache μνημών στα πολυπεξεργαστικά συστήματα.

1.2.2 Μέγιστη επιτάχυνση - Νόμος του Amdahl

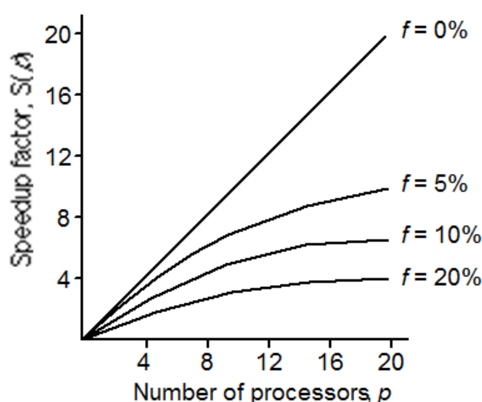
Σχεδόν σε όλους τους υπολογισμούς υπάρχει ένα τμήμα που είναι ακολουθιακό και ένα τμήμα το οποίο μπορεί να διαιρεθεί σε επιμέρους υποέργα. Έστω t_s ο συνολικός χρόνος της ακολουθιακής εκτέλεσης του υπολογισμού και έστω ft_s ο χρόνος εκτέλεσης του πρώτου τμήματος και $(1-f)t_s$ ο χρόνος εκτέλεσης του δεύτερου τμήματος. Σε ένα πολυεπεξεργαστή με p επεξεργαστές, το πρώτο τμήμα εξακολουθεί να έχει χρόνο εκτέλεσης ft_s, ενώ το δεύτερο τμήμα εκτελείται σε χρόνο $(1-f)t_s/p$ Άρα η επιτάχυνση για αυτό τον υπολογισμό δίνεται από την ακόλουθη σχέση:

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{p}{1+(p-1)f}.$$

Ο παραπάνω τύπος είναι γνωστός ως νόμος του Amdahl.



Σχήμα 1.2: Μέγιστη επιτάχυνση - Νόμος του Amdahl.



Σχήμα 1.3: Επιτάχυνση για διαφορετικό πλήθος επεξεργαστών.

Ακόμα και με άπειρο πλήθος επεξεργαστών, η μέγιστη επιτάχυνση είναι το πολύ $1/f$. Για παράδειγμα, με $f = 5\%$, η μέγιστη επιτάχυνση είναι 20, ανεξάρτητα του πλήθους των επεξεργαστών.

1.2.3 Παράδειγμα Υπεργραμμικής Επιτάχυνσης - Αλγόριθμοι Αναζήτησης

Σε ένα απλό αλγόριθμο που επιλύει ένα πρόβλημα αναζήτησης, ο χώρος των λύσεων εξετάζεται εξαντλητικά. Υποθέτουμε ο χώρος των λύσεων χωρίζεται σε p υποχώρους και αυτοί μοιράζονται στους p επεξεργαστές και κάθε ένας επεξεργαστής εκτελεί μία ανεξάρτητη αναζήτηση στον υποχώρο που του αντιστοιχεί. Έστω ότι η λύση βρίσκεται στο $x + 1$ υποχώρο και μάλιστα μπορεί να βρεθεί σε Δt βήματα ακολουθιακά σε αυτό τον υποχώρο. Η συνολική επιτάχυνση θα είναι:

$$S(p) = (xt_s/p + \Delta t)/\Delta t.$$

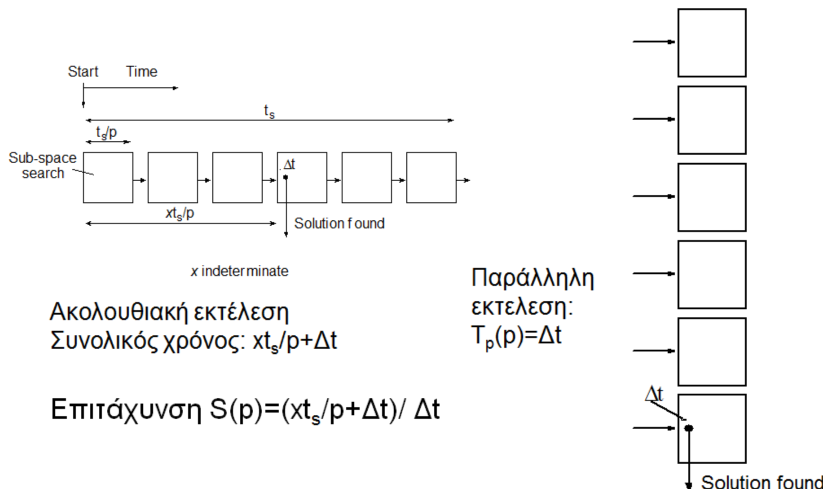
Η χειρότερη περίπτωση για τον ακολουθιακό αλγόριθμο συμβαίνει όταν η λύση βρίσκεται στο τελευταίο υποχώρο:

$$S(p) = (xt_s/p + \Delta t)/\Delta t \to \infty, \quad \text{καθώς } \Delta t \to 0.$$

Ελάχιστη επιτάχυνση θα έχουμε όταν η λύση βρίσκεται στο πρώτο υποχώρο:

$$S(p) = \Delta t/\Delta t = 1.$$

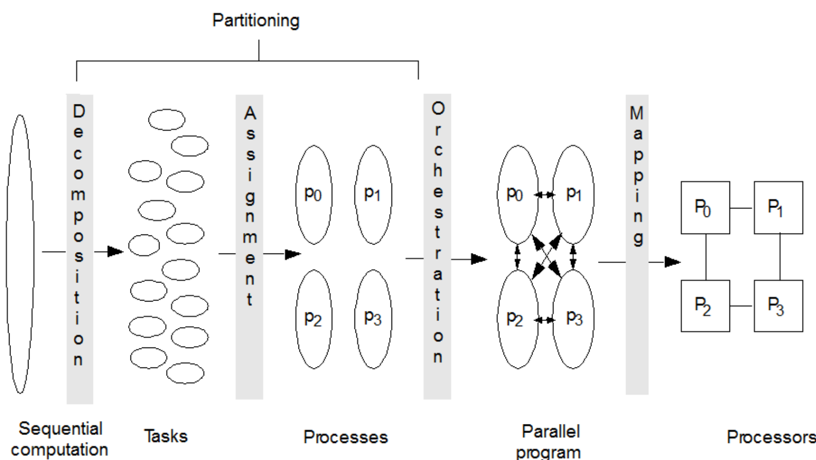
Στην πράξη, η επιτάχυνση εξαρτάται από τον υποχώρο που περιέχει τη λύση.



Σχήμα 1.4: Παράδειγμα Υπεργραμμικής Επιτάχυνσης - Αλγόριθμοι Αναζήτησης.

1.3 Παράλληλος Προγραμματισμός

Παράλληλο Πρόγραμμα ονομάσεται ένα πρόγραμμα που περιέχει δύο ή περισσότερες διεργασίες (processes), οι οποίες εργάζονται μαζί για την επίλυση του ίδιου προβλήματος. Η διεργασία (process) είναι ένα ακολουθιακό πρόγραμμα (ακολουθία εντολών). Οι πολλαπλές διεργασίες ενός παράλληλου προγράμματος συνεργάζονται επικοινωνώντας η μια με την άλλη. Η επικοινωνία πραγματοποιείται με τη χρήση κοινών ή διαμοιραζόμενων μεταβλητών (shared variables) ή μέσω αποστολής και παραλαβής μηνυμάτων (message passing).



Σχήμα 1.5: Στάδια δημιουργίας παράλληλου προγράμματος.

Τα στάδια δημιουργίας ενός παράλληλου προγράμματος είναι τα εξής:

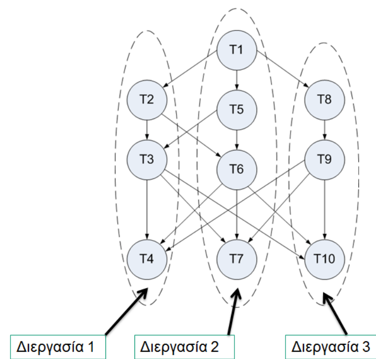
1. Ο ακολουθιακός υπολογισμός διαιρείται σε υποέργα.

2. Τα υποέργα μοιράζονται σε πλήθος διεργασιών.

3. Στις περισσότερες περιπτώσεις, ένα υποέργο χρειάζεται τα αποτελέσματα της επεξεργασίας ενός άλλου υποέργου. Αυτό απαιτεί συντονισμό μεταξύ των διεργασιών του παράλληλου προγράμματος. Η συνεργασία μεταξύ των διεργασιών απαιτεί επικοινωνία μεταξύ τους.

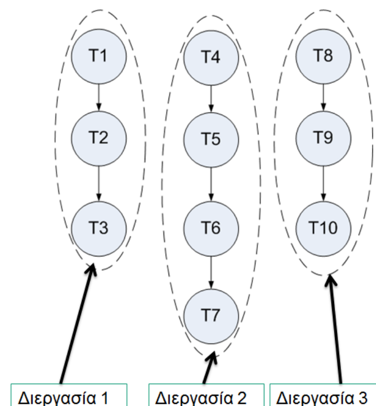
4. Το τελευταίο βήμα είναι η αντιστοίχηση των διεργασιών στους επεξεργαστές ενός πολυεπεξεργαστικού συστήματος. Κάθε επεξεργαστής εκτελεί τις διεργασίες που του έχουν ανατεθεί.

Κατά την παράλληλη εκτέλεση εργασιών, τελικός στόχος είναι η ολοκλήρωση όλων των υποέργων στον συντομότερο δυνατό χρόνο. Γενικά, το πρόβλημα είναι NP-complete.



Σχήμα 1.6: Παράλληλη εκτέλεση εργασιών. Κάθε ακμή μεταξύ των τριών διαμερίσεων δηλώνει επικοινωνία μεταξύ των αντίστοιχων διεργασιών.

Σε ορισμένα προβλήματα δεν υπάρχουν εξαρτήσεις μεταξύ των υποέργων. Σε αυτή την περίπτωση, έχουμε ιδανική επιτάχυνση (Embarrassingly parallel program).



Σχήμα 1.7: Ένα παράδειγμα εύκολου παραλληλισμού.

Υπάρχουν δύο τρόποι επικοινωνίας των επεξεργαστών ενός παράλληλου συστήματος:

- Η χρήση κοινής ή διαμοιραζόμενης μνήμης (shared memory).

- Η επικοινωνία των επεξεργαστών μέσω ανταλλαγής ή περάσματος μηνυμάτων (message passing).

Αντίστοιχα έχουμε:

- Συστήματα τα οποία διαθέτουν κοινό χώρο διευθύνσεων (shared dataspace): Πολυεπεξεργαστές • Συστήματα Κοινής Μνήμης (Shared-memory systems - multiprocessors).

- Συστήματα τα οποία υποστηρίζουν πέρασμα μηνυμάτων. Έχουμε δύο περιπτώσεις:

  { Πολυ-υπολογιστές - Συστήματα Κατανεμημένης Μνήμης (Distributed-memory systems - multicomputers).

  { Clusters (Networks of Workstations (NoWs) and Beowulf Clusters).

1.3.1 Πολυεπεξεργαστές (Multiprocessors)

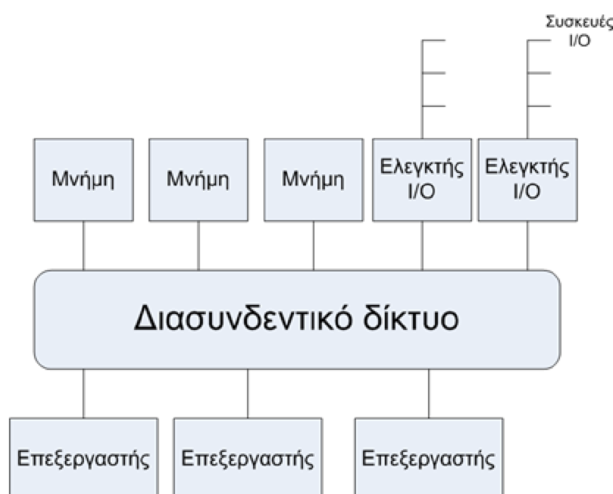
Αποτελούνται από έναν αριθμό επεξεργαστών οι οποίοι έχουν πρόσβαση στην ίδια κοινή μνήμη.

Υπάρχουν δύο κατηγορίες πολυεπεξεργαστών:

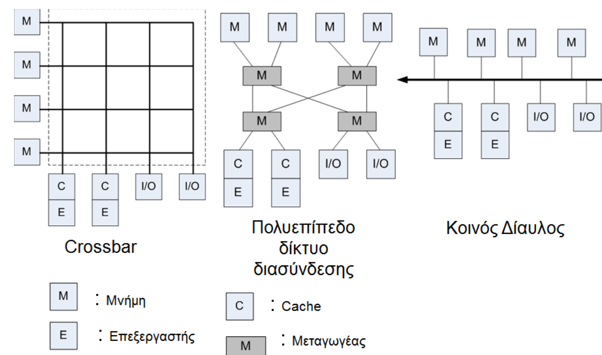
- Πολυεπεξεργαστές Ομοιόμορφης Προσπέλασης Μνήμης (Uniform Memory Access • UMA), στους οποίους η κοινή μνήμη είναι κεντρική.

- Πολυεπεξεργαστές Μη Ομοιόμορφης Προσπέλασης Μνήμης (Non-Uniform Memory Access • NUMA), στους οποίους η κοινή μνήμη είναι κατανεμημένη στους επεξεργαστές.

Οι επεξεργαστές κοινής μνήμης έχουν πρόσβαση σε μία κεντρική κοινή μνήμη μέσω ενός κεντρικού μηχανισμού προσπέλασης μνήμης. Η επικοινωνία των επεξεργαστών γίνεται μέσω των συμβατικών εντολών πρόσβασης μνήμης (load - store). Μέρος της εικονικής μνήμης των διεργασιών είναι κοινό δηλ. αντιστοιχεί σε κοινό χώρο της μνήμης.



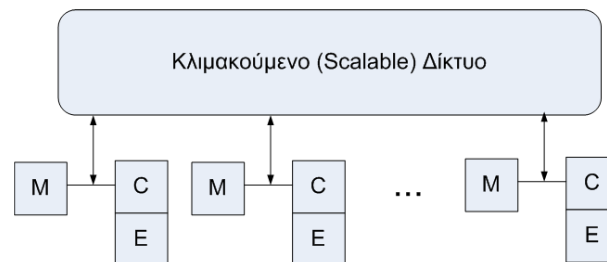
Σχήμα 1.8: Κοινή μνήμη.



Σχήμα 1.9: Τυπικά σχήματα διασύνδεσης μνημών • επεξεργαστών.

Τυπικά σχήματα διασύνδεσης μνημών • επεξεργαστών: Το δίκτυο crossbar παρέχει ξεχωριστή ζεύξη για κάθε ζεύγος επεξεργαστή μνήμη. Είναι λύση με υψηλό κόστος. Ο δίαυλος είναι η πιο οικονομική λύση αλλά δημιουργεί συχνά φαινόμενα συμφόρησης, αφού όλες οι επικοινωνίες πραγματοποιούνται μέσω του κοινού διαύλου. Μία ενδιάμεση λύση από πλευράς κόστους και απόδοσης είναι το πολυεπίπεδο δίκτυο διασύνδεσης.

Στους Πολυεπεξεργαστές Μη Ομοιόμορφης Προσπέλασης Μνήμης (NUMA), κάθε επεξεργαστής διαθέτει τη δική του μνήμη. Δίνεται η ψευδαίσθηση ότι υπάρχει κοινή διαμοιραζόμενη μνήμη. Όμως ο χρόνος για την ολοκλήρωση μιας εγγραφής ή ανάγνωσης διαφέρει σημαντικά ανάλογα αν η λειτουργία στη τοπική μνήμη ή σε μνήμη άλλου επεξεργαστή.



Σχήμα 1.10: Πολυεπεξεργαστές Μη Ομοιόμορφης Προσπέλασης Μνήμης (NUMA).

Προγραμματισμός Πολυεπεξεργαστών (Συστήματα Διαμοιραζόμενης Μνήμης): Ο προγραμματισμός σε τέτοια συστήματα γίνεται με τους ακόλουθους τρόπους:

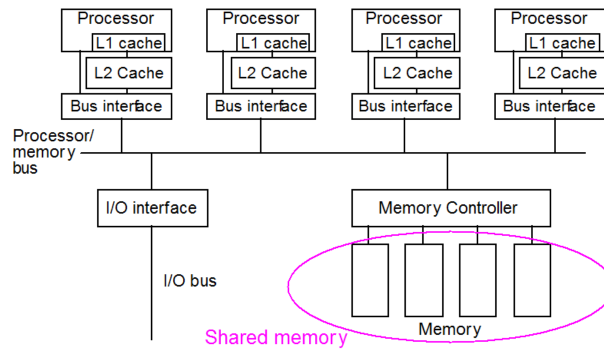
- Νήματα (Threads) • ο προγραμματιστής χωρίζει το πρόγραμμα σε ξεχωριστές παράλληλες ακολουθίες εντολών (threads). Κάθε νήμα μπορεί να προσπελάζει σφαιρικές μεταβλητές δηλωμένες έξω από κάθε νήμα.

  Παράδειγματα: Pthreads, Java threads, CUDA (γλώσσα προγραμματισμού των καρτών NVIDIA).

- Σε μία "κλασσική" γλώσσα προγραμματισμού εισάγονται οδηγίες προεπεξεργασίας (directives) προς το μεταγλωττιστή, οι οποίες δηλώνουν κοινές μεταβλητές και προσδιορίζουν παραλληλισμό.

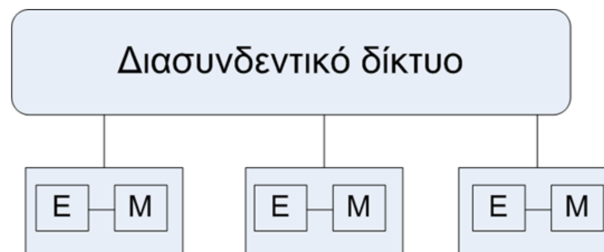
  Παράδειγμα: η OpenMP, η οποία αποτελεί πρότυπο για το προγραμματισμό πολυεπεξεργαστών.

- Ο προγραμματιστής γράφει σε μία "κλασσική" γλώσσα προγραμματισμού και στη συνέχεια ο μεταγλωττιστής μετατρέπει το πρόγραμμα σε παράλληλο. Γενικά, η αυτόματη παραλληλοποίηση είναι δύσκολο ζήτημα και δεν επιτυγχάνει πάντα αποδοτικές λύσεις



Σχήμα 1.11: Quad Pentium Shared Memory Multiprocessor.

Πολυ-υπολογιστές (Multicomputers) - Συστήματα Κατανεμημένης Μνήμης: Κάθε επεξεργαστής έχει τη δική του τοπική μνήμη. Δεν υπάρχει κοινή μνήμη στην οποία έχουν πρόσβαση όλοι οι επεξεργαστές. Η επικοινωνία των επεξεργαστών γίνεται με πέρασμα μηνυμάτων (message-passing) μέσω δικτύου διασύνδεσης.



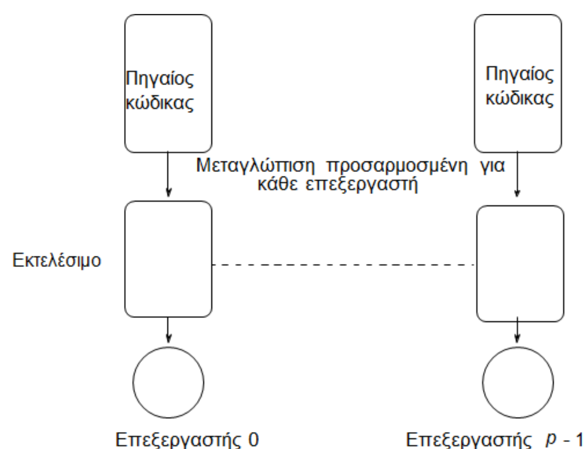
Σχήμα 1.12: Συστήματα Κατανεμημένης Μνήμης.

Κεφάλαιο 2

Προγραμματισμός με πέρασμα μηνυμάτων

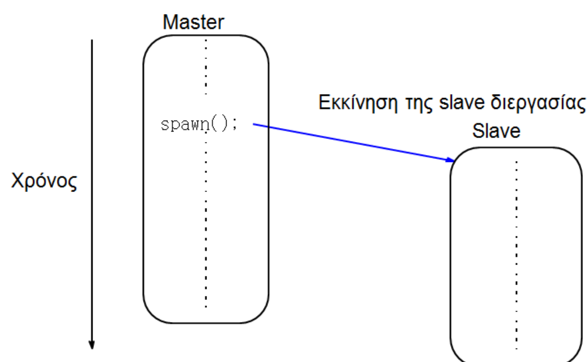
Στον προγραμματισμό με πέρασμα μηνυμάτων απαιτούνται δύο μηχανισμοί: Μία μέθοδος για τη δημιουργία διεργασιών που θα εκτελούνται σε διαφορετικούς υπολογιστές και μία μέθοδος για την αποστολή και παραλαβή μηνυμάτων.

Multiple program, multiple data (MPMD) model: Ξεχωριστός κώδικας γράφεται και μεταφράζεται για κάθε επεξεργαστή. Οι διεργασίες που εκτελούν τους διαφορετικούς κώδικες στους επεξεργαστές, μπορούν να δημιουργούνται δυναμικά από άλλες διεργασίες που ήδη τρέχουν σε άλλους επεξεργαστές Συνήθως οι διεργασίες έχουν σχέση master-slave. Μία διεργασία (master) αναλαμβάνει το συντονισμό όλου του υπολογισμού. Όλες οι υπόλοιπες διεργασίες (slaves) δημιουργούνται δυναμικά από τη master διαδικασία. Ουσιαστικά, οι slave διαδικασίες αναλαμβάνουν το κύριο όγκο επεξεργασίας και στέλνουν τα μερικά τους αποτελέσματα στη master διεργασία, η οποία τα συνδυάζει και ενδεχομένως τα ξαναστέλνει στις slave διεργασίες.



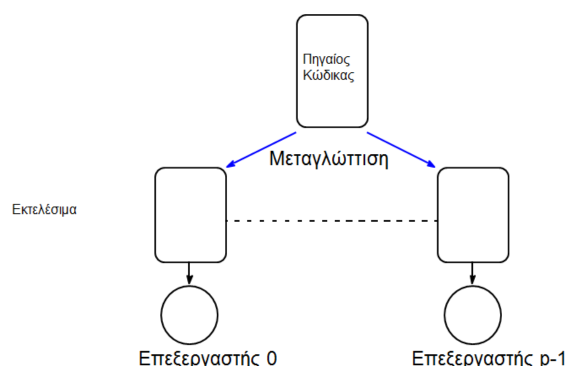
Σχήμα 2.1: Προγραμματισμός με πέρασμα μηνυμάτων.

Single Program Multiple Data (SPMD) model. Όλες οι διεργασίες εκτελούν το ίδιο πρόγραμμα. Με τη χρήση όμως εντολών ελέγχου και ανάλογα το αναγνωριστικό της διαδικασίας, διαφορετικά τμήματα κώδικα εκτελούνται από τις διεργασίες. Όλα τα εκτελέσιμα ξεκινούν από



Σχήμα 2.2: Multiple program, multiple data (MPMD) model.

την αρχή. Έχουμε δηλαδή στατική δημιουργία διεργασιών, όπου ο χρήστης θα πρέπει να έχει καθορίσει εξ' αρχής το πλήθος των διεργασιών (στατική δημιουργία διεργασιών).

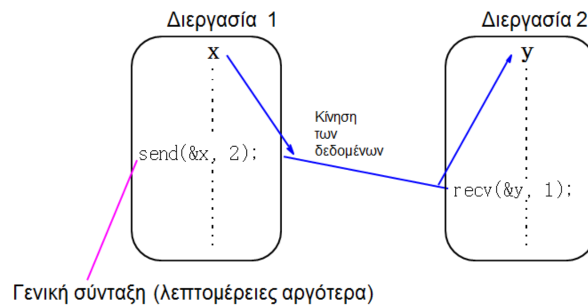


Σχήμα 2.3: Single program, multiple data (SPMD) model.

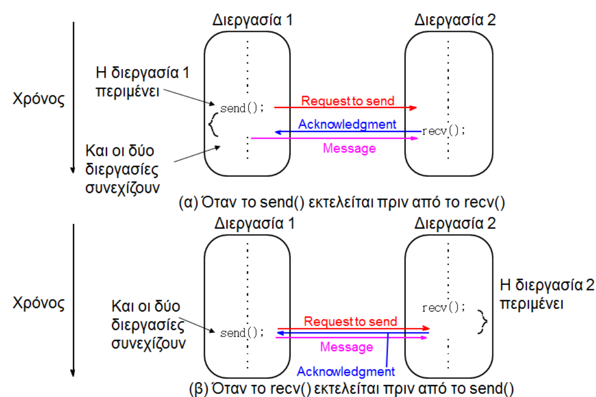
Η αποστολή ενός μηνύματος μεταξύ δύο διεργασιών γινεται με τη κλήση των συναρτήσεων send() και recv() της βιβλιοθήκης του MPI. Στο συγκεκριμένο παράδειγμα, η τιμή της τοπικής μεταβλητής x της διεργασίας 1 στέλνεται στη διεργασία 2 και αποθηκεύεται στη τοπική μεταβλητή της διεργασίας 2.

2.1 Σύγχρονο (Synchronous) πέρασμα μηνυμάτων

Οι συναρτήσεις που υλοποιούν σύγχρονο πέρασμα μηνυμάτων, επιστρέφουν μόνο αφότου έχει ολοκληρωθεί η μεταφορά του μηνύματος. Σύγχρονη ρουτίνα αποστολής μηνύματος: Αυτή η λειτουργία περιμένει μέχρι ολόκληρο το μήνυμα ληφθεί από τη διαδικασία παραλήπτη, πριν να αρχίσει την αποστολή νέου μηνύματος. Σύγχρονη ρουτίνα λήψης μηνύματος: Αυτή η λειτουργία περιμένει μέχρι το μήνυμα που αναμένει φθάσει. Ουσιαστικά, οι σύγχρονες συναρτήσεις εκτελούν δύο ενέργειες: Μεταφέρουν δεδομένα και συγχρονίζουν τις διαδικασίες.



Σχήμα 2.4: Βασικές ρουτίνες αποστολής λήψης μεταξύ δύο επεξεργαστών.



Σχήμα 2.5: Σύγχρονο send() και recv() χρησιμοποιώντας ένα πρωτόκολλο τριών σταδίων (3-way protocol).

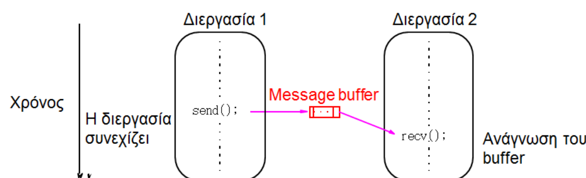
2.2 Ασύγχρονο (asynchronous) πέρασμα μηνυμάτων

Οι συναρτήσεις αυτές επιστρέφουν πριν ολοκληρωθεί η λειτουργία επικοινωνίας που εκτελούν. Απαιτείται συνήθως τοπική αποθήκευση των μηνυμάτων που είναι σε εκκρεμότητα για αποστολή ή λήψη. Γενικά, δεν συγχρονίζουν τις διεργασίες, αλλά επιτρέπουν σε αυτές να προχωρούν στην εκτέλεση του προγράμματός τους νωρίτερα. Θα πρέπει όμως να χρησιμοποιούνται με προσοχή. Υπάρχουν δύο κατηγορίες ασύγχρονων λειτουργιών:

- Blocking - επιστρέφουν αφότου οι τοπικές τους ενέργειες ολοκληρωθούν αν και η μεταφορά των μηνυμάτων μπορεί να μην έχει ολοκληρωθεί εκείνη τη χρονική στιγμή.

- Non-blocking • επιστρέφουν κατευθείαν μετά την κλήση τους. Βασική υπόθεση είναι ότι πριν να ολοκληρωθεί η μεταφορά του μηνύματος, η τοπική μνήμη που διατίθεται για τη μεταφορά δεδομένων δεν τροποποιείται από τις εντολές που ακολουθούν. Είναι ευθύνη του προγραμματιστή να εξασφαλίσει ότι δεν θα συμβεί το αντίθετο.

Οι ασύγχρονες blocking συναρτήσεις μπορούν να "εκφυλιστούν" σε σύγχρονες ρουτίνες. Μόλις οι τοπικές ενέργειες ολοκληρωθούν και η αποστολή του μηνύματος έχει αρχίσει, ο αποστολέας μπορεί να συνεχίσει στην εκτέλεση των επόμενων εντολών. Όμως, οι buffers έχουν μόνο πεπερασμένο μέγεθος και έτσι μπορεί να προκύψει η κατάσταση όπου η διαδικασία αποστολής σταματά διότι όλος ο διαθέσιμος χώρος στους buffers έχει εξαντληθεί. Σε αυτή την περίπτωση,

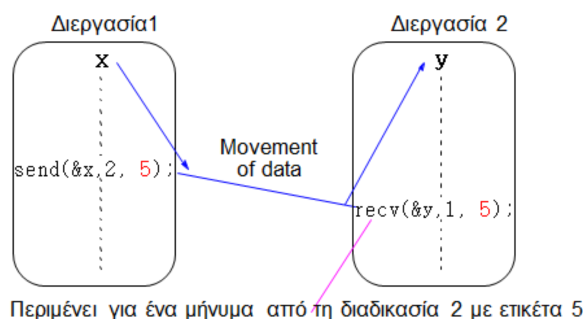


Σχήμα 2.6: Πώς οι ρουτίνες περάσματος μηνυμάτων επιστρέφουν πριν η μεταφορά μηνυμάτων ολοκληρωθεί.

η διαδικασία αποστολής θα περιμένει μέχρι να ελευθερωθεί χώρος. Με άλλα λόγια, η ρουτίνα συμπεριφέρεται ως μία σύγχρονη συνάρτηση.

2.2.1 Ετικέτα μηνύματος

Η ετικέτα μηνύματος χρησιμοποιείται για να διαφοροποιήσει μεταξύ των διαφορετικών τύπων δεδομένων που πρόκειται να σταλούν. Η ετικέτα μεταφέρεται με το μήνυμα. Αν δεν υπάρχει απαίτηση για συγκεκριμένο τύπο δεδομένων, μία ετικέτα μπαλαντέρ (wild card message tag) χρησιμοποιείται, έτσι ώστε το recv() να λάβει τα δεδομένα οποιουδήποτε send(). Για παράδειγμα, για να στείλουμε τα δεδομένα που είναι αποθηκευμένα στη μεταβλητή x, με ετικέτα μηνύματος 5 από τη διαδικασία 1 στη διαδικασία 2 και καταχώρηση των ληφθέντων δεδομένων στη μεταβλητή y, θα έχουμε:



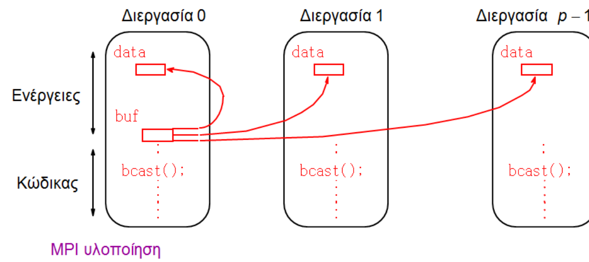
Σχήμα 2.7: Ετικέτα μηνύματος.

Οι ομαδικές ρουτίνες περάσματος μηνυμάτων στέλνουν μηνύματα σε μία ομάδα διεργασιών ή λαμβάνουν μηνύματα από μία ομάδα διεργασιών. Θα μπορούσαν να υλοποιηθούν με μία σειρά από "σημείο-σε-σημείο" διαδικασιών. Συνήθως όμως, υψηλότερες επιδόσεις επιτυγχάνονται όταν υλοποιούνται κατευθείαν, χωρίς την εκτέλεση πολλαπλών "σημείο-σε-σημείο" διαδικασιών.

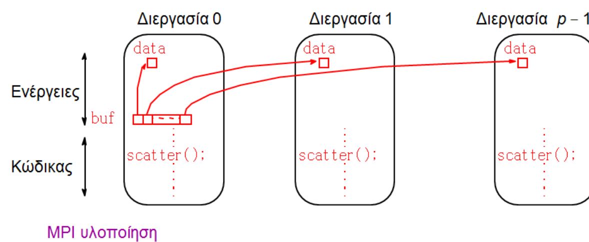
Μία άλλη τεχνική περάσματος μηνυμάτων είναι αυτή της εκπομπής (broadcast) , κατά την οποία γίνεται αποστολή του ίδιου μηνύματος σε όλες τις διεργασίες. Μία παραλλαγή της εκπομπής είναι το multicast, όπου το ίδιο μήνυμα παραλαμβάνεται από ένα συγκεκριμένο υποσύνολο διεργασιών που έχει από πριν προσδιορισθεί.

2.3 Διασπορά - Scatter

Η διεργασία ρίζα μοιράζει τα στοιχεία ενός πίνακα σε ξεχωριστές διεργασίες. Συγκεκριμένα, τα περιεχόμενα της i-οστής θέσης του πίνακα στέλνονται στην i-οστή διεργασία.



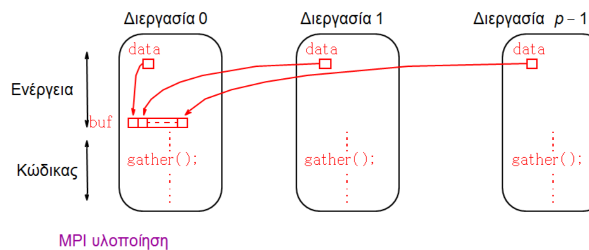
Σχήμα 2.8: Broadcast.



Σχήμα 2.9: Scatter.

2.4 Συγκέντρωση - Gather

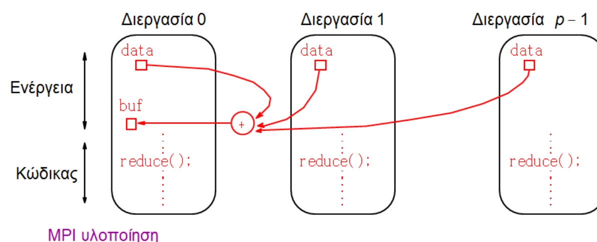
Σε αυτή τη λειτουργία, μία διεργασία (η διεργασία ρίζα) συλλέγει ξεχωριστά δεδομένα από ένα σύνολο διεργασιών. Έχει ουσιαστικά το αντίστροφο αποτέλεσμα από τη διασπορά.



Σχήμα 2.10: Gather.

2.5 Μείωση - Reduce

Η λειτουργία αυτή είναι μία λειτουργία συγκέντρωσης όπου στα δεδομένα που συγκεντρώνονται εκτελείται μία συγκεκριμένη αριθμητική-λογική λειτουργία. Για παράδειγμα, οι τιμές συγκεντρώνονται σε μία διεργασία (διεργασία ρίζα) και στη συνέχεια αθροίζονται από αυτή τη διεργασία:



Σχήμα 2.11: Παράδειγμα μείωσης: Οι τιμές συγκεντρώνονται σε μία διεργασία (διεργασία ρίζα) και στη συνέχεια αθροίζονται από αυτή τη διεργασία.

2.6 MPI (Message Passing Interface)

Η βιβλιοθήκη ΜΠΙ είναι ένα πρότυπο παράλληλου προγραμματισμού συστημάτων κατανεμημένης μνήμης και είναι αποτέλεσμα της σύμπραξης ερευνητικών κέντρων και επιχειρήσεων. Ορίζει μόνο τις συναρτήσεις (παράμετροι εισόδου, τιμές επιστροφής κτλ.) και όχι πώς αυτές υλοποιούνται. Υπάρχουν πολλές "ελεύθερες" υλοποιήσεις της βιβλιοθήκης.

Σκοπίμως δεν ορίζεται ο τρόπος με τον οποίο δημιουργούνται οι διεργασίες και εκτελούνται. Οι λεπτομέρειες καθορίζονται από τη συγκεκριμένη υλοποίηση. Στην έκδοση 1 του MPI, μόνο στατικές διεργασίες υποστηριζόταν. Όλες οι διεργασίες θα πρέπει να οριστούν πριν την εκτέλεση του παράλληλου προγράμματος και αρχίσουν όλες μαζί. Αρχικά επίσης, υποστηριζόταν μόνο το μοντέλο υπολογισμού SPMD. Και το μοντέλο MPMD μπορεί να συνδυαστεί με το στατικό τρόπο δημιουργίας διεργασιών - κάθε πρόγραμμα που θα εκτελεστεί ορίζεται εξ' αρχής.

2.6.1 Communicators

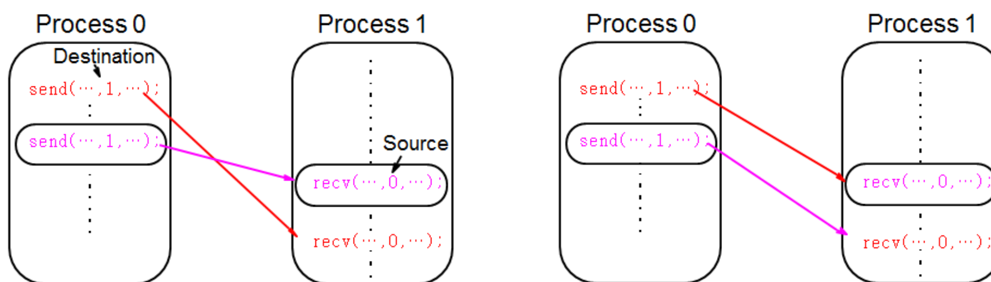
Οι communicators ορίζουν την εμβέλεια μίας λειτουργίας επικοινωνίας. Σε κάθε διεργασία που ανήκει στον ίδιο communicator αντιστοιχεί ένα μοναδικό αναγνωριστικό (rank). Αρχικά, όλες οι διεργασίες ανήκουν σε έναν "παγκόσμιο" communicator, τον MPI_COMM_WORLD. Επίσης κάθε διεργασία έχει ένα μοναδικό αναγνωριστικό, ένα αριθμό που κυμαίνεται από 0 μέχρι $p-1$, όπου p είναι το πλήθος των διεργασιών του παράλληλου προγράμματος. Στη συνέχεια, άλλοι communicators μπορούν να οριστούν σε υποομάδες των p διεργασιών.

```
main (int argc, char *argv[])
{
MPI_Init(&argc, &argv);
.
.
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */
if (myrank == 0)
master();
else
slave();
.
.
MPI_Finalize();
}
```

Υπάρχει μία διεργασία master (myrank=0) και μία ή περισσότερες διεργασίες slave (myrank > 0) Αν και το ίδιο πρόγραμμα τρέχει σε όλες τις διεργασίες, εντούτοις οι διεργασίες εκτελούν διαφορετικούς υπολογισμούς ανάλογα με τη τιμή του αναγνωριστικού κάθε διεργασίας. Η συνάρτηση MPI_Comm_rank επιστρέφει το αναγνωριστικό της διεργασίας που την εκτελεί.

2.6.2 Πιθανά σενάρια εκτέλεσης των συναρτήσεων send και receive

Τα δεδομένα από το πρώτο (δεύτερο) send της διεργασίας 0 θα πρέπει να παραληφθούν από το δεύτερο (πρώτο) recv της δεύτερης διεργασίας. Για να λυθεί το πρόβλημα, κάθε μήνυμα θα πρέπει να έχει μία μοναδική ετικέτα (MESSAGE_TAG) και κάθε recv πρέπει να δηλώνει την ετικέτα του μηνύματος που θα λάβει.



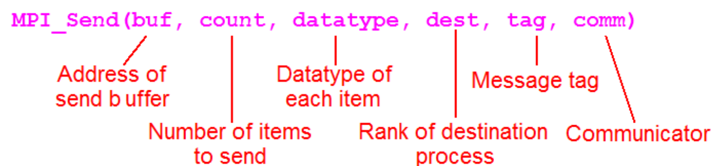
Σχήμα 2.12: Πιθανά σενάρια εκτέλεσης των συναρτήσεων send και receive.

2.6.3 MPI - Επικοινωνία Σημείο σε Σημείο (Point-to-Point)

Οι βασικές ρουτίνες send και receive υλοποιούν τις επικοινωνίες Σημείο-σε-Σημείο. Υπάρχουν και blocking και non-blocking διαδικασίες send και recv. Οι blocking ρουτίνες επιστρέφουν όταν η τοπική επεξεργασία έχει ολοκληρωθεί, δηλαδή όταν η θέση που κρατά το μήνυμα μπορεί να χρησιμοποιηθεί ξανά ή να αλλαχθεί χωρίς να επηρεαστεί η αποστολή ή λήψη του μηνύματος. Όταν ένα blocking send επιστρέφει δεν σημαίνει ότι το μήνυμα έχει ληφθεί από τον παραλήπτη, απλά η διεργασία είναι ελεύθερη να συνεχίσει με την εκτέλεση των επόμενων εντολών στο πρόγραμμα χωρίς να επηρεαστεί η εν εξελίξει αποστολή μηνύματος.

Παράμετροι του blocking send:

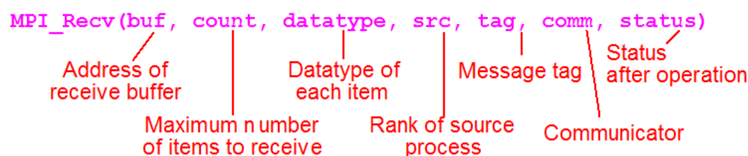
- buf: ο χώρος που περιέχει τα δεδομένα που θα σταλούν.
- count: το πλήθος των δεδομένων που θα σταλούν.
- datatype: ο τύπος των δεδομένων που θα σταλούν: MPI_INT για ακεραίους, MPI_CHAR για χαρακτήρες, MPI_FLOAT για αριθμούς κινητής υποδιαστολής.
- dest: το αναγνωριστικό της διεργασίας παραλήπτη του μηνύματος.
- tag: η ετικέτα με την οποία θα σταλεί το μήνυμα.
- comm: ο communicator στην εμβέλεια του οποίου θα εκτελεσθεί η αποστολή του μηνύματος.



Σχήμα 2.13: Παράμετροι του blocking send.

Παράμετροι του blocking receive:

- buf: ο χώρος που θα δεχθεί τα δεδομένα που θα ληφθούν.
- count: το μέγιστο πλήθος των δεδομένων που θα ληφθούν.
- datatype: ο τύπος των δεδομένων που θα ληφθούν: MPI_INT για ακεραίους, MPI_CHAR για χαρακτήρες, MPI_FLOAT για αριθμούς κινητής υποδιαστολής.
- src: το αναγνωριστικό της διεργασίας αποστολέα του μηνύματος.
- tag: η ετικέτα του μηνύματος που θα λάβει το παραλήπτης.
- comm: ο communicator στην εμβέλεια του οποίου θα εκτελεσθεί η λήψη του μηνύματος.



Σχήμα 2.14: Παράμετροι του blocking receive.

Ακολουθεί παράδειγμα αποστολής ενός ακεραίου αποθηκευμένο στη τοπική μεταβλητή x της διεργασίας 0 στη διεργασία 1. Τα δεδομένα λαμβάνονται και αποθηκεύονται στη τοπική μεταβλητή x της διεργασίας 1.

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */
if (myrank == 0) {
int x;
MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
int x;
MPI_Recv(&x, 1, MPI_INT, 0,msgtag,MPI_COMM_WORLD,status);
}
```

2.6.4 Nonblocking MPI συναρτήσεις

Υπάρχουν οι ακόλουθες non-blocking συναρτήσεις:

```
MPI_Isend(buf,count,datatype,dest,tag,comm,request)
MPI_Irecv(buf,count,datatype,source,tag,comm, request)
```

- Nonblocking send - MPI_Isend(). Η λειτουργία αυτή επιστρέφει κατευθείαν πριν ακόμα ο αποθηκευτικός χώρος του μηνύματος που θα σταλεί να ελευθερωθεί και επομένως να είναι έτοιμος για τροποποίηση.
- Nonblocking receive - MPI_Irecv(). Η λειτουργία αυτή θα επιστρέψει αμέσως ακόμα και όταν δεν υπάρχει μήνυμα για παραλαβή.

Η ολοκλήρωση των συναρτήσεων μπορούν να διαπιστωθούν με τις συναρτήσεις MPI_Wait() και MPI_Test(). Η MPI_Wait() περιμένει μέχρι η λειτουργία να ολοκληρωθεί και στη συνέχεια επιστρέφει. Η MPI_Test() επιστρέφει αμέσως και θέτει μία λογική μεταβλητή που προσδιορίζει αν η εκκρεμής λειτουργία είχε ολοκληρωθεί τη στιγμή που έγινε ο έλεγχος. Για να ελέγξουμε μία συγκεκριμένη διαδικασία, θα πρέπει να περάσουμε την παράμετρο request που έχει πάρει τιμή από την MPI_Send()/MPI_Recv() στις συναρτήσεις MPI_Wait(), MPI_Test().

Ακολουθεί παράδειγμα αποστολής ενός ακεραίου x από την διεργασία 0 στη διεργασία 1. Η διεργασία 0 επιτρέπεται να συνεχίσει χωρίς να περιμένει την ολοκλήρωση της αποστολής μηνύματος. Θα περιμένει όμως στην MPI_Wait.

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
int x;
MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
compute();
MPI_Wait(req1, status);
} else if (myrank == 1) {
int x;
MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

2.6.5 Συλλεκτική (Collective) Επικοινωνία

Οι λειτουργίες συλλογικής επικοινωνίας δρουν στο σύνολο των διεργασιών που περιλαμβάνονται σε ένα communicator. Οι λειτουργίες αυτές δεν χρησιμοποιούν ετικέτες μηνυμάτων. Οι κύριες συλλεκτικές λειτουργίες επικοινωνίας είναι:

- MPI_Bcast(): Εκπομπή από μία συγκεκριμένη διεργασία (διεργασία ρίζα) σε όλες τις άλλες διεργασίες.
- MPI_Gather(): Συλλογή δεδομένων από μία ομάδα διεργασιών στη διεργασία ρίζα.
- MPI_Scatter(): Διασκόρπιση των δεδομένων της διεργασίας ρίζας σε μία ομάδα διεργασιών.
- MPI_Alltoall(): Κάθε διεργασία στέλνει διαφορετικά δεδομένα σε κάθε άλλη διεργασία.
- MPI_Reduce(): Εκτελεί το συνδυασμό (π.χ. άθροισμα, εύρεση ελάχιστου/μέγιστου) όλων των τιμών που δίνουν οι διεργασίες και το αποτέλεσμα αποθηκεύεται στη διεργασία ρίζα.
- MPI_Reduce_scatter(): Συνδυάζει τις τιμές των διεργασιών όπως η Reduce αλλά στη συνέχεια διασκορπίζει το αποτέλεσμα σε όλες τις διεργασίες
- MPI_Scan(): Εκτελεί το υπολογισμό prefix στα δεδομένα των διεργασιών.

Ακολουθεί παράδειγμα συλλογής δεδομένων από μία ομάδα διεργασιών στη διεργασία 0, χρησιμοποιώντας δυναμικά καταχωρημένη μνήμη στη διεργασία ρίζα:

```
int data[10]; /*data to be gathered from processes*/
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
MPI_Comm_size(MPI_COMM_WORLD, &grp_size); /*find group size*/
buf = (int *)malloc(grp_size*10*sizeof (int)); /*allocate memory*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_COMM_WORLD);
```

Η MPI_Gather() συλλέγει δεδομένα από τις διεργασίες συμπεριλαμβανομένης και της ρίζας.

2.6.6 Barrier (Φράγμα)

Όπως σε όλα τα συστήματα περάσματος μηνυμάτων, έτσι και το MPI παρέχει ένα τρόπο για συγχρονισμό των διεργασιών. Αυτό σημαίνει ότι όταν η διεργασία κατά την εκτέλεση του προγράμματος φτάσει στη εντολή Barrier περιμένει μέχρι όλες οι άλλες διεργασίες να εκτελέσουν επίσης την εντολή Barrier. Στη συνέχεια, όλες οι διεργασίες προχωρούν στην εκτέλεση των επόμενων εντολών.

Παράδειγμα προγράμματος MPI:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv)
{
int myid, numprocs;
int data[MAXSIZE], i, x, low, high, myresult, result;
char fn[255];
char *fp;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if (myid == 0) {  /* Open input file and initialize data */
strcpy(fn,getenv("HOME"));
strcat(fn,"/MPI/rand_data.txt");
if ((fp = fopen(fn,"r")) == NULL) {
printf("Can't open the input file: %s\n\n", fn);
exit(1);
}
for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
}
MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */
x = n/nproc; /* Add my portion Of data */
low = myid * x;
high = low + x;
for(i = low; i < high; i++)
myresult += data[i];
printf("I got %d from %d\n", myresult, myid); /* Compute global sum */
MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
if (myid == 0) printf("The sum is %d.\n", result);
MPI_Finalize();
}
```

2.7 Αποτίμηση απόδοσης παράλληλων προγραμμάτων

Χρόνος εκτέλεσης ακολουθιακού υπολογισμού t_s: Υπολογίζεται με τη μέτρηση των βημάτων του καλύτερου ακολουθιακού αλγόριθμου.

Χρόνος εκτέλεσης παράλληλου υπολογισμού, t_p: Επιπροσθέτως του πλήθους των υπολογιστικών βημάτων t_{comp}, χρειάζεται να υπολογίσουμε την χρονική επιβάρυνση λόγω των επικοινωνιών μεταξύ των διεργασιών του παράλληλου προγράμματος t_{comm}:

$$t_p = t_{comp} + t_{comm}.$$

2.7.1 Χρόνος Υπολογισμού

Όπως αναφέρθηκε, ο χρόνος υπολογισμού προκύπτει από τη μέτρηση των βημάτων υπολογισμού. Όταν περισσότερες από μία διαδικασίες εκτελούνται ταυτόχρονα, μετρούμε τα βήματα της πιο σύνθετης διαδικασίας. Γενικά, ο χρόνος υπολογισμού είναι μία συνάρτηση του μεγέθους των δεδομένων n και του πλήθους των επεξεργαστών p, δηλαδή

$$t_{comp} = f(n, p).$$

Συχνά ο υπολογισμός σπάει σε επιμέρους στάδια, τα οποία χωρίζονται από γύρους επικοινωνίας όπου οι επεξεργαστές ανταλλάσουν αποτελέσματα από τα προηγούμενα στάδια επικοινωνίας. Έτσι, ο συνολικός χρόνος υπολογισμού μπορεί να γραφεί ως:

$$t_{comp} = t_{comp1} + t_{comp2} + t_{comp3} + \cdots$$

Η ανάλυση συνήθως γίνεται με την υπόθεση ότι όλοι οι επεξεργαστές είναι ίδιοι και λειτουργούν με την ίδια ταχύτητα.

2.7.2 Χρόνος Επικοινωνίας

Πολλοί παράγοντες επηρεάζουν το χρόνο επικοινωνίας όπως η δομή του διασυνδετικού δικτύου καθώς και της συμφόρεσης που μπορεί να παρουσιάζει. Σαν μία πρώτη προσέγγιση, χρησιμοποιούμε τον τύπο

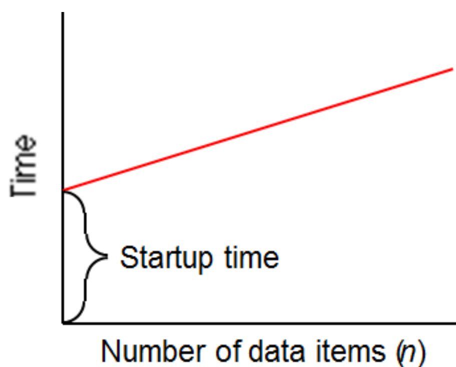
$$t_{comm} = t_{startup} + nt_{data},$$

όπου $t_{startup}$ είναι ο χρόνος εκκίνησης και ουσιαστικά είναι ο χρόνος να σταλεί ένα μήνυμα όταν δεν περιέχει δεδομένα. Συνήθως ο χρόνος αυτός θεωρείται σταθερός. Η παράμετρος t_{data} είναι ο χρόνος μεταφοράς για να σταλεί μία λέξη. Επίσης στο παραπάνω τύπο, n είναι το πλήθος των λέξεων που στέλνονται κατά την επικοινωνία μεταξύ δύο διεργασιών.

Ο τελικός χρόνος επικοινωνίας είναι το άθροισμα των χρόνων επικοινωνίας όλων των μηνυμάτων που εστάλησαν από μία διεργασία, δηλαδή

$$t_{comm} = t_{comm1} + t_{comm2} + t_{comm3} + \cdots$$

Τυπικά, τα μοτίβα επικοινωνίας όλων των διεργασιών είναι ίδια και επιπλέον θεωρείται ότι συμβαίνουν ταυτόχρονα. Έτσι, για τον υπολογισμό του συνολικού χρόνου επικοινωνίας, συνήθως αρκεί να εξετάσουμε μία μόνο από τις παράλληλες διεργασίες. Τόσο ο χρόνος εκκίνησης $t_{startup}$, όσο και ο χρόνος μετάδοσης δεδομένων t_{data} εκφράζονται σε μονάδες χρόνου του ενός βήματος υπολογισμού και έτσι μπορούμε να αθροίσουμε τους χρόνους t_{comp} και t_{comm} μαζί, για να λάβουμε το χρόνο εκτέλεσης t_p.



Σχήμα 2.15: Χρόνος Επικοινωνίας.

2.7.3 Δείκτες Απόδοσης

Με γνωστά τα t_s, t_comp και t_{comm}, μπορούμε να υπολογίσουμε τους παρακάτω δείκτες απόδοσης:

$$\text{Speedup factor} \ = \frac{t_s}{t_p} = \frac{t_s}{t_{comp} + t_{comm}},$$

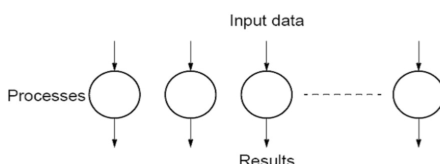
$$\text{Computation/communication ratio} \ = \frac{t_{comp}}{t_{comm}}.$$

Και οι δύο δείκτες είναι συναρτήσεις του πλήθους των επεξεργαστών p, και των πλήθους των δεδομένων n. Και οι δύο δείκτες δείχνουν πόσο κλιμακούμενη είναι η παράλληλη λύση με την αύξηση του πλήθους των επεξεργαστών και του μεγέθους του προβλήματος. Επίσης, ο λόγος Υπολογισμός/Επικοινωνία δείχνει την επιβάρυνση λόγω επικοινωνίας στο συνολικό χρόνο λύσης με την αύξηση του πλήθους των επεξεργαστών και του μεγέθους του προβλήματος.

Κεφάλαιο 3

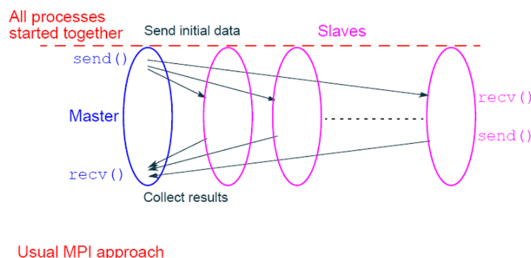
Πολύ εύκολες παραλληλοποιήσεις

Υπάρχουν παραδείγματα υπολογισμών οι οποίοι μπορούν με προφανή τρόπο να διαιρεθούν σε τελείως ανεξάρτητα έργα. Καθένα από αυτά τα έργα μπορούν να εκτελεστούν από ξεχωριστές διεργασίες (επεξεργαστές)



Σχήμα 3.1: Ανεξάρτητα έργα.

Οι διεργασίες είτε δεν επικοινωνούν καθόλου ή η επικοινωνία τους είναι ελάχιστη. Κάθε διεργασία εκτελεί τα έργα της χωρίς αλληλεπίδραση με τις άλλες διεργασίες.



Σχήμα 3.2: Ανεξάρτητα έργα.

Συνήθως αυτού του είδους οι υπολογισμοί, υλοποιούνται με την τεχνική master-slave. Αρχικά, η διεργασία master μοιράζει τα δεδομένα εισόδου σε ένα πλήθος slave διεργασιών. Κάθε διεργασία slave υλοποιεί ένα τμήμα του υπολογισμού χωρίς να επικοινωνεί με άλλες διεργασίες. Με την ολοκλήρωση του υπολογισμού που τις έχει ανατεθεί, κάθε διεργασία slave στέλνει το τοπικό αποτέλεσμα στη διεργασία master. Η διεργασία master συνδυάζει όλα τα αποτελέσματα από τις slave διεργασίες και υπολογίζει το τελικό αποτέλεσμα.

Πολύ εύκολες παραλληλοποιήσεις (Embarrassingly Parallel Computations):

- Επεξεργασία εικόνας χαμηλού επιπέδου
- Σύνολο Mandelbrot

- Υπολογισμοί Monte Carlo

3.1 Επεξεργασία εικόνας χαμηλού επιπέδου

Πολλές λειτουργίες από την περιοχή της επεξεργασίας εικόνας είναι τοπικές, δηλαδή το αποτέλεσμα σε κάθε pixel της εικόνας προκύπτει ως συνάρτηση της τρέχουσας τιμής του pixel ή και των γειτονικών pixels. Αν η εικόνα διαιρεθεί σε περιοχές και κάθε περιοχή ανατεθεί σε ξεχωριστή διεργασία, οι διεργασίες μπορούν να επεξεργαστούν τις τοπικές τους περιοχές χωρίς να επικοινωνούν μεταξύ τους.

Ακολουθούν κάποιες συχνές λειτουργίες επεξεργασίας εικόνας σε χαμηλό επίπεδο.

Μερικοί γεωμετρικοί μετασχηματισμοί:

- **Ολίσθηση.** Το αντικείμενο ολισθαίνει κατά Dx κατά μήκος του άξονα x και κατά Dy κατά μήκος του άξονα y:

$$x¢ = x + Dx$$
$$y¢ = y + Dy$$

  όπου x και y είναι οι αρχικές συντεταγμένες και $x¢$, $y¢$ είναι οι νέες συντεταγμένες.

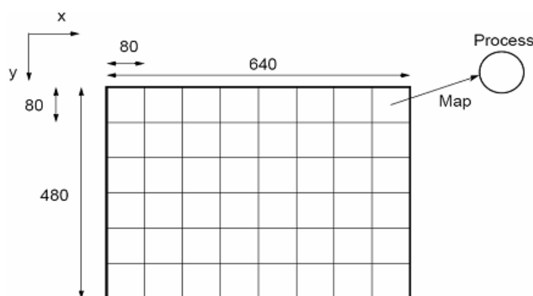
- **Κλιμάκωση.** Τα αντικείμενο κλιμακώνεται με συντελεστή Sx στον άξονα των x και με συντελεστή Sy στον άξονα των y:

$$x¢ = xSx$$
$$y¢ = ySy$$

- **Περιστροφή.** Το αντικείμενο περιστρέφεται κατά μία γωνία q γύρω από την αρχή των αξόνων:

$$x¢ = xcosq + ysinq$$
$$y¢ = -xsinq + ycosq$$

Όλοι αυτοί οι υπολογισμοί μπορούν εύκολα να παραλληλοποιηθούν χωρίς οι παράλληλες διεργασίες να επικοινωνούν μεταξύ τους.



Σχήμα 3.3: Διαμοιρασμός μιας εικόνας στις διεργασίες.

Η εικόνα διαιρείται σε ορθογώνιες περιοχές (80×80) και κάθε διεργασία επεξεργάζεται την περιοχή που τις αντιστοιχεί. Εναλλακτικά, η εικόνα μπορεί να διαιρεθεί σε οριζόντιες περιοχές (strips).

3.2 Το σύνολο Mandelbrot

Το σύνολο Mandelbrot είναι ένα σύνολο σημείων στο μιγαδικό χώρο τα οποία είναι ημι-σταθερά, δηλαδή αυξάνονται και μειώνονται χωρίς να υπερβαίνουν ένα συγκεκριμένο όριο, και υπολογίζονται επαναληπτικά με βάση τη συνάρτηση

$$z_{k+1} = z_k^2 + c,$$

όπου z_{k+1} είναι ο μιγαδικός αριθμός $z = a+bi$ που προκύπτει μετά από την $(k+1)$-οστή επανάληψη και c είναι ένας μιγαδικός αριθμός, ο οποίος δίνει τη θέση του σημείου στο επίπεδο των μιγαδικών. Η αρχική τιμή του z είναι μηδέν. Οι επαναλήψεις συνεχίζονται μέχρι το μέγεθος του z να υπερβεί το 2 ή το πλήθος των επαναλήψεων φτάσει ένα συγκεκριμένο όριο. Το μέγεθος $||z||$ του z είναι το μήκος του διανύσματος και δίνεται από την ακόλουθη σχέση:

$$||z|| = \sqrt{a^2 + b^2}.$$

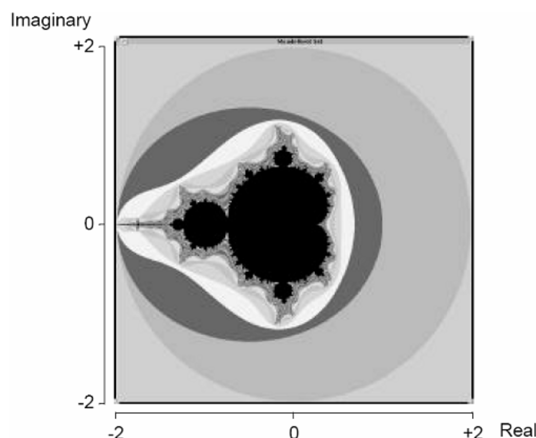
Ακολουθεί η ακολουθιακή διαδικασία για τον υπολογισμό της τιμής σε ένα σημείο. Επιστρέφει το πλήθος των επαναλήψεων.

```
structure complex {
    float real;
    float imag;
};

int cal_pixel(complex c){
    int count, max;
    complex z;
    float temp, lengthsq;
    max = 256;
    z.real = 0; z.imag = 0;
    count = 0;               /* number of iterations */
    do {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        count++;
    } while ((lengthsq < 4.0) && (count < max));
    return count;
}
```

Το αποτέλεσμα της εκτέλεσης του προηγούμενου κώδικα φαίνεται στο σχήμα. Οι συντεταγμένες κάθε pixel είναι το πραγματικό και φανταστικό τμήμα του μιγαδικού c, ο οποίος περνάει ως όρισμα στη συνάρτηση cal_pixel(). Το πλήθος των επαναλήψεων που επιστρέφονται καθορίζουν και το χρώμα του αντίστοιχου pixel.

3.2.1 Παράλληλη υλοποίηση του υπολογισμού του συνόλου Mandelbrot

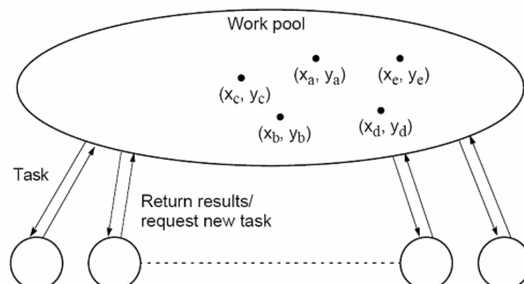
Στατική καταχώρηση έργων: Σε αυτή την προσέγγιση, η περιοχή διαιρείται σε ένα σταθερό πλήθος περιοχών και κάθε διεργασία αναλαμβάνει τον υπολογισμό σε μία τέτοια περιοχή. Δεν είναι καλύτερη προσέγγιση για το συγκεκριμένο πρόβλημα, αφού διαφορετικές περιοχές



Σχήμα 3.4: Το σύνολο Mandelbrot.

απαιτούν διαφορετικό πλήθος επαναλήψεων και επομένως χρόνο. Αυτό έχει ως αποτέλεσμα κάποιοι επεξεργαστές να εργάζονται ενώ άλλοι να είναι αδρανείς, έχοντας τελειώσει νωρίτερα την εργασία τους.

Δυναμική καταχώρηση έργων: Χρειάζεται να επιτευχθεί εξισορρόπηση φορτίου σε όλες τις διεργασίες. Η δυναμική καταχώρηση έργων διατηρεί μία «δεξαμενή» με τα προς εκτέλεση έργα. Κάθε διεργασία αναλαμβάνει την εκτέλεση του επόμενου διαθέσιμου έργου. Όταν μία διεργασία ολοκληρώσει το έργο που έχει αναλάβει, ζητά το επόμενο έργο από τη δεξαμενή και η διαδικασία αυτή επαναλαμβάνεται μέχρι η δεξαμενή να «αδειάσει» από έργα. Συνήθως, η master διεργασία αναλαμβάνει τη διαχείριση της δεξαμενής έργων.



Σχήμα 3.5: Δυναμική καταχώριση έργων για τον υπολογισμό του συνόλου Mandelbrot.

Στο συγκεκριμένο υπολογισμό, κάθε έργο αντιστοιχεί στον υπολογισμό που γίνεται σε ένα pixel της εικόνας. Λόγω του μικρού όγκου υπολογισμού (fine-grained υπολογισμός) που αναλαμβάνει κάθε φορά μία διεργασία, ζητάει συχνά επιπλέον έργα για να εκτελέσει, με αποτέλεσμα τη συχνή επικοινωνία μεταξύ master και slave διαδικασιών. Για αυτό το λόγο, τα έργα που μοιράζονται στις slave διεργασίες περιλαμβάνουν συνήθως περισσότερους υπολογισμούς. Συγκεκριμένα, κάθε έργο είναι οι υπολογισμοί που πρέπει να γίνουν για τα pixels μίας γραμμής της εικόνας.

Ακολουθεί ψευδοκώδικας για τον παράλληλο υπολογισμό του συνόλου Mandelbrot.

```
Master
count=0;
```

```
row=0;
for (k=0; k< num_proc; k++)
   send(&row, Pk, data_tag);
   count++
   row++
}
do {
   recv(&slave, &r, color, PANY, result_tag);
   count--;
   if (row < disp_height) {
       send( &row, Pslave, data_tag);
       row++;
       count++;
   }
   else
       send( &row, Pslave,terminator_tag);
   display(r, color);
} while (count >0)
```

```
Slave
recv(y, Pmaster, source_tag);
while (source_tag == data_tag) {
c.imag = imag_min + ((float) y * scale_imag);
for (x=0; x < disp_width; x++) {
   c.real = real_min + ((float) x *scale_real);
   color[x]=cal_pixel(c);
}
send(&x, &y, color, Pmaster, result_tag);
recv(&y, Pmaster, source_tag);
}
```

3.3 Μέθοδοι Monte Carlo

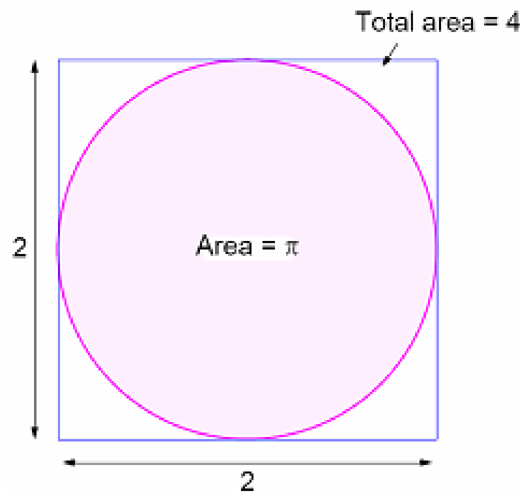
Οι μέθοδοι Monte Carlo χρησιμοποιούν τυχαίες επιλογές στους υπολογισμούς. Βρίσκουν εφαρμογή στη λύση αριθμητικών υπολογισμών και προβλημάτων φυσικής. Οι υπολογισμοί που αντιστοιχούν στις τυχαίες επιλογές μπορούν να γίνουν ανεξάρτητα ο ένας από τον άλλο και επομένως και οι μέθοδοι Monte Carlo είναι υπολογισμοί με πολύ εύκολη παραλληλοποίηση.

Παράδειγμα ; υπολογισμός του π. Ένα παράδειγμα Monte Carlo μεθόδου είναι ο υπολογισμός του π. Συγκεκριμένα, αν υποθέσουμε ένα κύκλο ακτίνας ίσης με 1 εντός ενός τετραγώνου 2×2, τότε ο λόγος της εμβαδού του κύκλου προς το εμβαδό του τετραγώνου είναι:

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}.$$

Εκτελούμε ένα τυχαίο πείραμα επιλέγοντας σημεία μέσα στο τετράγωνο με τυχαίο τρόπο. Στη συνέχεια, μετρούμε το πλήθος των περιπτώσεων που το τυχαία επιλεγμένο σημείο είναι εντός

του κύκλου. Το ποσοστό f των σημείων εντός κύκλου θα συγκλίνει στο $\pi/4$, μετά από αρκετές επαναλήψεις του πειράματος, δηλαδή $f = \pi/4$. Επιλύοντας τον παραπάνω τύπο ως προς π, λαμβάνουμε μία προσεγγιστική τιμή για το π.



Σχήμα 3.6: Παράδειγμα ; υπολογισμός του π.

3.3.1 Υπολογισμός Ολοκληρώματος με τη μέθοδο Monte Carlo

Για τον υπολογισμό του ολοκληρώματος $\int_a^b f(x)dx$, υπολογίζουμε τη τιμή της συνάρτησης f σε τυχαία σημεία $x_1, x_2, \ldots, x_N$, στο διάστημα $[a, b]$. Στη συνέχεια, το ολοκλήρωμα θα είναι ίσο με

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} f(x_i)(b-a).$$

Η μέθοδος Monte Carlo είναι πολύ χρήσιμη, εάν δεν μπορεί να βρεθεί το ολοκλήρωμα της συνάρτησης με κάποια αριθμητική μέθοδο, όταν για παράδειγμα περιέχει ένα μεγάλο αριθμό μεταβλητών. Για παράδειγμα, για τον υπολογισμό του ολοκληρώματος

$$\int_{x_1}^{x_2} (x^2 - 3x)dx$$

ο ακολουθιακός κώδικας θα είναι:

```
sum = 0;
for (i = 0; i < N; i++) {            /* N random samples */
    xr = rand_v(x1, x2);             /* generate next random value */
    sum = sum + xr * xr - 3 * xr;    /* compute f(xr) */
}
area = (sum / N) * (x2 - x1);
```

Στη παράλληλη υλοποίηση της μεθόδου Monte Carlo, κάθε slave διεργασία αναλαμβάνει τον υπολογισμό της συνάρτησης σε ένα ή περισσότερα σημεία και στη συνέχεια αθροίζουν τις τιμές της συνάρτησης. Η master διεργασία αναλαμβάνει να αθροίσει τα μερικά αθροίσματα που τις στέλνουν οι slave διεργασίες σύμφωνα με το παραπάνω τύπο.

Ακολουθεί ψευδοκώδικας για την παράλληλη υλοποίηση της ολοκλήρωσης Monte Carlo.

```
Master
n=N/num_slaves
for (i=0; i<N/n; i++){
   for (j=0; j<n; j++)
  xr[j]=rand();
   recv(PANY,req_tag,Psource);
   send(xr, &n, Psource,compute_tag);
}
for (i=0; i<num_slaves;i++){
   recv(Pi, req_tag);
   send(Pi, stop_tag);
}
sum = 0;
reduce_add(&sum, Pgroup);

Slave
sum=0;
send(Pmaster, req_tag);
recv(xr, &n, Pmaster, source_tag);
while (source_tag == compute_tag){
   for (i=0; i < n; i++)
   sum = sum +xr[i]*xr[i] -3*xr[i];
   send(Pmaster, req_tag);
   recv(xr, &n, Pmaster, source_tag);
}
reduce_add(&sum, Pgroup);
```

Κεφάλαιο 4

Στρατηγικές Διαμέρισης (Partitioning) και Διαίρει και Βασίλευε (Divide-and-Conquer)

Στην τεχνική της διαμέρισης, το αρχικό πρόβλημα διαιρείται σε μικρότερα προβλήματα. Συνήθως γίνεται διαμέριση των δεδομένων εισόδου και κάθε διεργασία αναλαμβάνει ένα τμήμα δεδομένων. Όπως είναι γνωστό, στην τεχνική Διαίρει και Βασίλευε, το πρόβλημα διαιρείται σε υποπροβλήματα της ίδιας μορφής με το αρχικό. Κάθε υποπρόβλημα μπορεί να διαιρεθεί με αναδρομικό τρόπο σε ακόμα μικρότερα προβλήματα, μέχρι να προκύψουν προβλήματα που η λύση τους προκύπτει κατευθείαν.

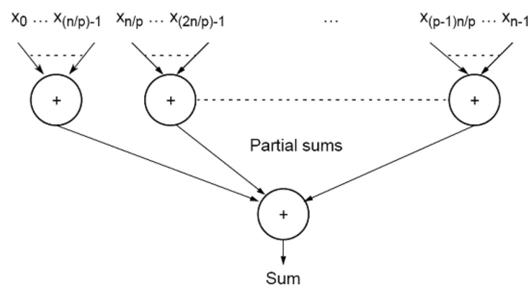
Για παράδειγμα, στην άθροιση των στοιχείων ενός πίνακα, μπορούμε να χωρίσουμε τον πίνακα εισόδου σε δύο υποπίνακες και στη συνέχεια να λύσουμε αναδρομικά το πρόβλημα της άθροισης στους δύο υποπίνακες. Το τελικό άθροισμα προκύπτει από το άθροισμα των μερικών αθροισμάτων των δύο υποπινάκων. Η αναδρομική διαδικασία της Διαίρει και Βασίλευε στρατηγικής μπορεί εύκολα να υλοποιηθεί παράλληλα, διότι ξεχωριστές διεργασίες μπορούν να αναλάβουν τα προβλήματα που προκύπτουν από την αναδρομική διαίρεση.

Παραδείγματα με εφαρμογή στρατηγικών Διαμέρισης και Διαίρει και Βασίλευε

- Λειτουργίες σε ακολουθίες αριθμών όπως η άθροιση.
- Μερικοί αλγόριθμοι ταξινόμησης συγκεκριμένα αλγόριθμοι που μπορούν να σχεδιαστούν με αναδρομή.
- Εύρεση ολοκληρώματος συναρτήσεων.
- Το πρόβλημα N-body.

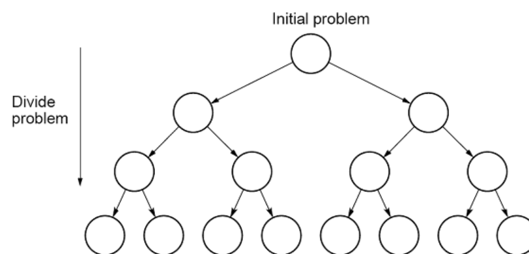
4.1 Παράδειγμα διαμέρισης ; Άθροισμα των στοιχείων μίας λίστας

Τα στοιχεία της λίστας χωρίζονται σε p υπολίστες, όπου p είναι το πλήθος των διεργασιών που θα εκτελέσουν το παράλληλο πρόγραμμα. Η διεργασία P_i, όπου $i = 0, 1, \ldots, p-1$, αναλαμβάνει να αθροίσει τα στοιχεία $x_{i*n/p} \cdots x_{(i+1)*n/p-1}$ και στη συνέχεια στέλνει το αποτέλεσμα της τοπικής άθροισης s_i στη master διεργασία, η οποία αθροίζει όλα τα μερικά αθροίσματα s_i και υπολογίζει το τελικό αποτέλεσμα.



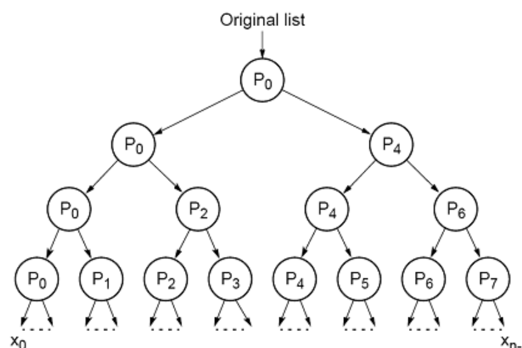
Σχήμα 4.1: Άθροισμα των στοιχείων μίας λίστας.

4.2 Τεχνική Διαίρει και Βασίλευε



Σχήμα 4.2: Η τεχνική Διαίρει και Βασίλευε.

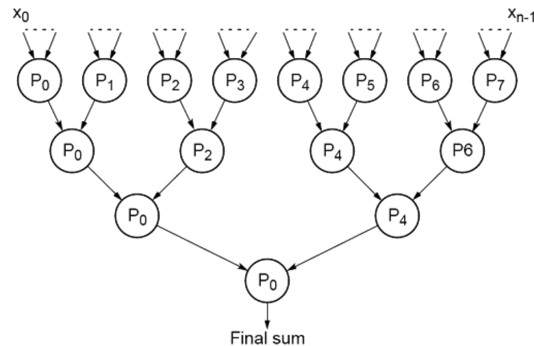
Η τεχνική διαίρει και βασίλευε μπορεί να αναπαρασταθεί με τη βοήθεια ενός δένδρου. Σε κάθε επίπεδο, κάθε πρόβλημα διαιρείται σε δύο υποπροβλήματα, μέχρι να φτάσουμε στο επίπεδο των φύλλων, όπου η λύση των υποπροβλημάτων είναι προφανής. Για παράδειγμα, στο πρόβλημα της άθροισης των στοιχείων μίας λίστας, το επίπεδο των φύλλων περιέχει μόνο στοιχειώδη προβλήματα, συγκεκριμένα άθροιση δύο στοιχείων.



Σχήμα 4.3: Άθροισμα των στοιχείων μίας λίστας.

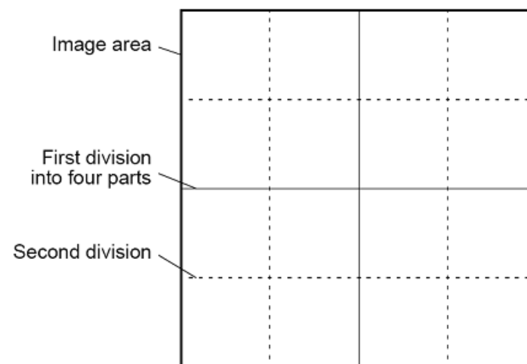
Η τεχνική διαίρει και βασίλευε μπορεί εύκολα να υλοποιηθεί παράλληλα. Στο πρόβλημα της άθροισης n στοιχείων $x_1, x_2, \ldots, x_{n-1}$, αρχικά η διαδικασία P_0 διαθέτει όλη τη λίστα των στοιχείων. Στη συνέχεια, διαιρεί την λίστα σε δύο υπολίστες και δίνει τη δεύτερη υπολίστα στη διεργασία P_4. Στη συνέχεια, οι διεργασίες P_0 και P_4 διαιρούν τις υπολίστες τους σε δύο μικρότερες υπολίστες και δίνουν τη δεύτερη υπολίστα στις διεργασίες P_2 και P_6. Αυτή η

διαδικασία επαναλαμβάνεται, μέχρι κάθε διεργασία να μείνει με $n/8$ στοιχεία της αρχικής λίστας.



Σχήμα 4.4: Άθροισμα των στοιχείων μίας λίστας.

Στη συνέχεια, κάθε διεργασία βρίσκει το άθροισμα των $n/8$ στοιχείων που διαθέτει. Κατόπιν, οι μισές διεργασίες στέλνουν τα μερικά τους αθροίσματα στις άλλες μισές. Οι διεργασίες αυτές αθροίζουν το δικό τους μερικό άθροισμα με αυτό που λαμβάνουν και στη συνέχεια η παραπάνω διαδικασία επαναλαμβάνεται στις τέσσερις εναπομείνασες διεργασίες, μέχρι το τελικό άθροισμα να προκύψει στη διεργασία P_0.

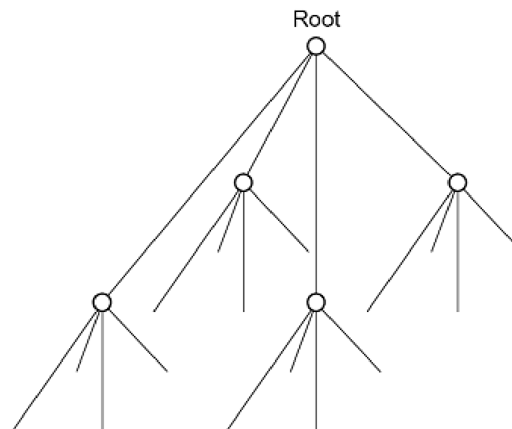


Σχήμα 4.5: Διαδοχική διαίρεση της εικόνας σε 4 περιοχές.

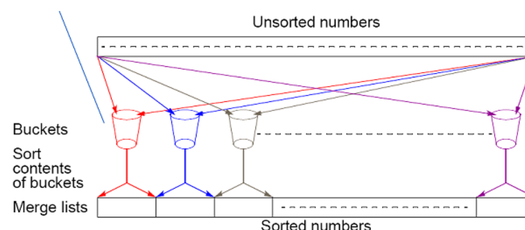
Στην τεχνική διαίρει και βασίλευε, το αρχικό πρόβλημα μπορεί να διαιρείται σε περισσότερα από δύο υποπροβλήματα σε κάθε βήμα. Μία συχνή περίπτωση προκύπτει στην εφαρμογή της διαίρει και βασίλευε τεχνικής στην επεξεργασία εικόνας. Σε αυτές τις περιπτώσεις εφαρμογών, σε κάθε βήμα η εικόνα διαιρείται σε 4 ίσου μεγέθους περιοχές. Το αντίστοιχο δέντρο που περιγράφει τον υπολογισμό είναι το τετραδικό δέντρο (quadtree).

4.3 Bucketsort

Ο αλγόριθμος bucket sort είναι ένας αλγόριθμος ταξινόμησης, όπου η βασική υπόθεση είναι ότι τα n στοιχεία του πίνακα που θα ταξινομηθεί είναι στο διάστημα $[0, a-1]$. Στη συνέχεια ορίζονται m "κάδοι" ("buckets") και κάθε κάδος αποθηκεύει στοιχεία σε μία συγκεκριμένη περιοχή. Πιο συγκεκριμένα, ο κάδος i, όπου $i = 0, 1, \ldots, m-1$, περιέχει στοιχεία στη περιοχή $[ia/m, (i+1)a/m - 1]$. Τα στοιχεία σε κάθε κάδο ταξινομούνται με τη χρήση ενός ακολουθιακού αλγόριθμου ταξινόμησης. Στη συνέχεια, οι ταξινομημένες λίστες των κάδων συγχωνεύονται σε



Σχήμα 4.6: Τετραδικό δένδρο.



Σχήμα 4.7: Bucketsort.

μία ταξινομημένη λίστα. Η συγχώνευση είναι εύκολη, τοποθετώντας στη τελική λίστα πρώτα τα περιεχόμενα του κάδου 0 μετά του κάδου 1 κοκ. Η συνολική πολυπλοκότητα του ακολουθιακού αλγορίθμου είναι $O(n \log(n/m))$, εφόσον τα στοιχεία του πίνακα κατανέμονται ομοιόμορφα στο διάστημα $[0, a-1]$.

4.3.1 Παράλληλη υλοποίηση του bucket sort - Απλή προσέγγιση

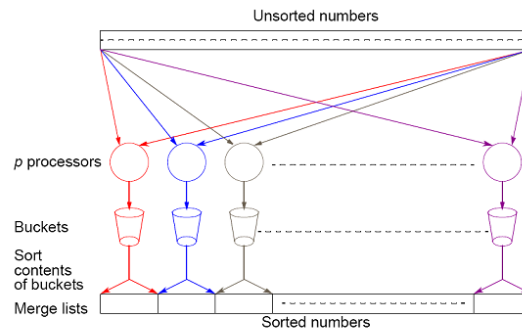
Κάθε διεργασία αναλαμβάνει την ταξινόμηση ενός κάδου ($p = m$). Κάθε διεργασία πρέπει να ελέγχει όλα τα στοιχεία του πίνακα εισόδου προκειμένου να απομονώσει τα στοιχεία που προορίζονται για τον κάδο του. Αυτό έχει ως αποτέλεσμα να γίνονται πολλοί άχρηστοι υπολογισμοί. Εναλλακτικά, κάθε διεργασία μπορεί να αφαιρεί από τον αρχικό πίνακα τα στοιχεία που θα τοποθετηθούν στο κάδο της. Έτσι, οι υπόλοιπες διεργασίες δεν θα εξετάσουν αυτά τα στοιχεία.

4.3.2 Παράλληλη υλοποίηση του bucket sort - Καλύτερη παράλληλη λύση

Η πιο αποδοτική παράλληλη υλοποίηση για το bucket sort διαιρεί τον αρχικό πίνακα σε p περιοχές και κάθε διεργασία αναλαμβάνει μία από αυτές τις περιοχές. Η i-οστή περιοχή ($i = 0, 1, \ldots, p-1$) περιέχει τα στοιχεία $x_{in/p}, \ldots, x_{(i+1)n/p-1}$ του αρχικού πίνακα.

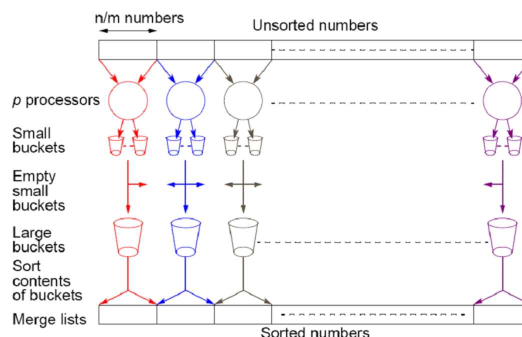
Κάθε διεργασία διατηρεί p "μικρούς" κάδους και χωρίζει τα στοιχεία της περιοχής του σε αυτούς τους μικρούς κάδους, ακολουθώντας τη λογική του bucket sort, δηλαδή ο κάδος i ($i = 0, 1, \ldots, p-1$) περιέχει στοιχεία στη περιοχή $[ia/p, (i+1)a/p - 1]$.

Στη συνέχεια, οι μικροί κάδοι αδειάζουν σε p τελικούς κάδους, όπου ο τελικός κάδος i ($i = 0, \ldots, p-1$) περιέχει στοιχεία στη περιοχή [ια/π,(ι+1)α/π-1]. Κάθε διεργασία αναλαμβάνει



Σχήμα 4.8: Bucketsort-Παράλληλη υλοποίηση.

έναν από αυτούς τους κάδους. Πιο συγκεκριμένα, κάθε διεργασία στέλνει τα περιεχόμενα του i-οστού μικρού κάδου στην i-οστή διεργασία, η οποία έχει υπΆ ευθύνη της τον τελικό κάδο i. Στη συνέχεια, κάθε διεργασία ταξινομεί τα περιεχόμενα του τελικού της κάδου και κατόπιν τα ταξινομημένα περιεχόμενα των τελικών κάδων συγχωνεύονται στην τελική λίστα. Σε αυτή την επικοινωνία, κάθε διεργασία στέλνει διαφορετικά δεδομένα σε κάθε άλλη διεργασία. Αυτό το μοτίβο επικοινωνίας είναι γνωστό ως all-to-all broadcast.



Σχήμα 4.9: Bucketsort-Παράλληλη υλοποίηση (καλύτερη λύση).

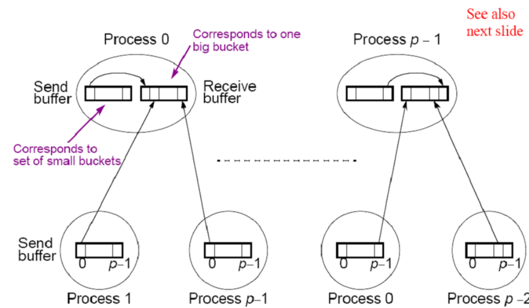
Η μεταφορά των δεδομένων στην all-to-all επικοινωνία μπορεί να ειδωθεί ως αναστροφή ενός πίνακα. Συγκεκριμένα, κάθε γραμμή αυτού του πίνακα αντιστοιχεί στα δεδομένα μίας διεργασίας και η all-to-all επικοινωνία μεταφέρει τις γραμμές αυτού του πίνακα στις στήλες.

4.4 Αριθμητική ολοκλήρωση με τη χρήση ορθογωνίων

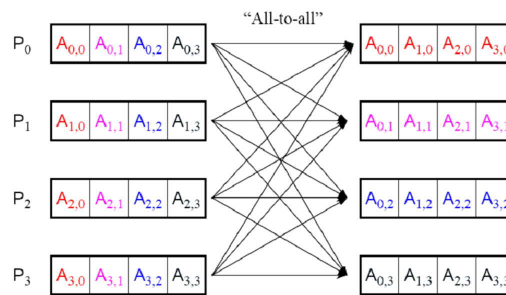
Για τον υπολογισμό του ολοκληρώματος $\int_a^b f(x)dx$, χωρίζουμε το διάστημα ολοκλήρωσης $[a, b]$ σε διαστήματα μήκους δ. Στη συνέχεια, για κάθε υποδιάστημα $[p, q]$, προσεγγίζουμε το ολοκλήρωμα της συνάρτησης με το εμβαδό του ορθογωνίου που έχει πλάτος δ και ύψος ίσο με $f((p + q)/2)$. Το συνολικό ολοκλήρωμα προκύπτει κατά προσέγγιση από το άθροισμα των εμβαδών αυτών των ορθογωνίων.

4.5 Αριθμητική ολοκλήρωση με τη χρήση τραπεζίων

Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε τραπέζια αντί για ορθογώνια. Και στις δύο περιπτώσεις, ο υπολογισμός στα διάφορα υποδιαστήματα μπορεί να μοιραστεί ισομερώς στις



Σχήμα 4.10: All-to-all broadcast. Κάθε διεργασία στέλνει διαφορετικά δεδομένα σε κάθε άλλη διεργασία.



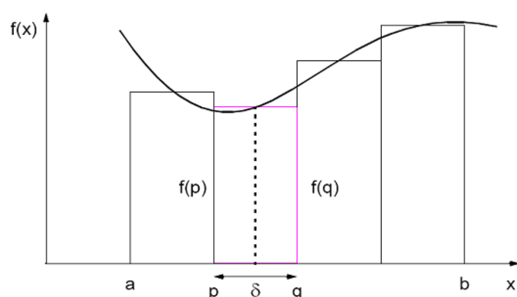
Σχήμα 4.11: All-to-all broadcast ως αναστροφή πίνακα.

διάφορες διεργασίες. Επειδή το πλήθος των υποδιαστημάτων είναι εκ των προτέρων γνωστό, μπορούμε να ακολουθήσουμε στατική καταχώριση έργων στις διεργασίες. Το πρόβλημα με αυτό τον τρόπο προσέγγισης ολοκληρώματος είναι ότι, σε περιπτώσεις συναρτήσεων που δεν έχουν ομαλό σχήμα, η προσέγγιση δεν είναι καλή.

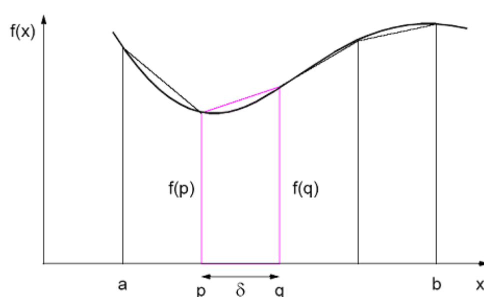
4.6 Προσαρμοζόμενος Τετραγωνισμός (Adaptive Quadrature)

Στην τεχνική αυτή, η λύση προσαρμόζεται στο σχήμα της γραφικής παράστασης. Τα υποδιαστήματα συνεχώς διαιρούνται μέχρι η προσέγγιση να είναι ικανοποιητική. Συγκεκριμένα, σε κάθε βήμα ελέγχεται αν το εμβαδό της μεγαλύτερης από τις δύο περιοχές A, B, C (C στο σχήμα) πλησιάζει το άθροισμα των εμβαδών των δύο άλλων περιοχών ($A + B$). Επειδή δεν είναι εκ των προτέρων γνωστό το πλήθος των τελικών διαστημάτων που προκύπτει από τις διαδοχικές διαιρέσεις, απαιτείται δυναμική καταχώριση των υπολογισμών στις διαθέσιμες διεργασίες.

Μία πιθανή λύση είναι αρχικά το διάστημα ολοκλήρωσης να διαιρεθεί σε ίσου μήκους διαστήματα και να ανατεθεί ένα διάστημα σε κάθε διεργασία. Κάθε διεργασία εκτελεί τον παραπάνω έλεγχο, και στην περίπτωση διαίρεσης, η διεργασία κρατάει το ένα υποδιάστημα και επιστρέφει το δεύτερο στη δεξαμενή των έργων (work pool). Αυτή η διαδικασία επαναλαμβάνεται μέχρι η προσέγγιση να είναι ικανοποιητική. Στη συνέχεια η διεργασία υπολογίζει το εμβαδό στο συγκεκριμένο διάστημα και το αποτέλεσμα επιστρέφεται στη master διεργασία. Στη συνέχεια, η διεργασία αναλαμβάνει το επόμενο διαθέσιμο διάστημα από τη δεξαμενή έργων και η παραπάνω διαδικασία επαναλαμβάνεται. Η master διεργασία υπολογίζει το συνολικό ολοκλήρωμα αθροίζοντας τα αποτελέσματα που στέλνουν οι slave διεργασίες.



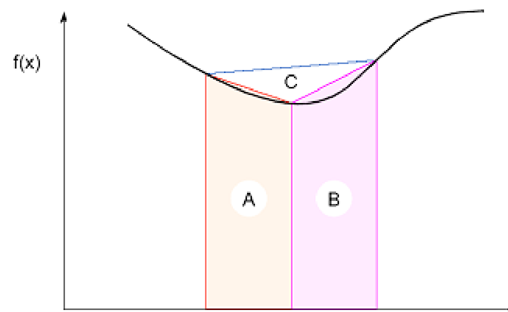
Σχήμα 4.12: Αριθμητική ολοκλήρωση με τη χρήση ορθογωνίων.



Σχήμα 4.13: Αριθμητική ολοκλήρωση με τη χρήση τραπεζίων.

4.7 Υπολογισμός της τιμής του π

```
/*********************************************************************************
pi_calc.cpp calculates value of pi and compares with actual
value (to 25digits) of pi to give error. Integrates function f(x)=4/(1+x^2).
July 6, 2001 K. Spry CSCI3145
*********************************************************************************/
#include <math.h> //include files
#include <iostream.h>
#include "mpi.h"

void printit();                                    //function prototypes
int main(int argc, char *argv[])
{
double actual_pi = 3.141592653589793238462643;
                                                   //for comparison later
int n, rank, num_proc, i;
double temp_pi, calc_pi, int_size, part_sum, x;
char response = 'y', resp1 = 'y';
MPI::Init(argc, argv);                              //initiate MPI

num_proc = MPI::COMM_WORLD.Get_size();
rank = MPI::COMM_WORLD.Get_rank();
```



Σχήμα 4.14: Προσαρμοζόμενος Τετραγωνισμός.

```
if (rank == 0) printit();             /* I am root node, print out welcome */

while (response == 'y') {
if (resp1 == 'y') {
if (rank == 0) {            /*I am root node*/
cout <<"__________________________________" <<endl;
cout <<"\nEnter the number of intervals: (0 will exit)" << endl;
cin >> n;}
} else n = 0;

MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0);     //broadcast n
if (n==0) break; //check for quit condition
else {
int_size = 1.0 / (double) n;                           //calcs interval size
part_sum = 0.0;

for (i = rank + 1; i <= n; i += num_proc)
 {                                                                //calcs partial sums
x = int_size * ((double)i - 0.5);
part_sum += (4.0 / (1.0 + x*x));
}
temp_pi = int_size * part_sum;
                                      //collects all partial sums computes pi

MPI::COMM_WORLD.Reduce(&temp_pi,&calc_pi, 1, MPI::DOUBLE, MPI::SUM, 0);

if (rank == 0) {                                                     /*I am server*/
cout << "pi is approximately " << calc_pi
<< ". Error is " << fabs(calc_pi - actual_pi)
<< endl
<<"_______________________________________"
<< endl;
}
}                                                                          //end else
if (rank == 0) { /*I am root node*/
cout << "\nCompute with new intervals? (y/n)" << endl; cin >> resp1;
```

```
}
}//end while
MPI::Finalize();                                              //terminate MPI
return 0;
}                                                                   //end main


//functions
void printit()
{
cout << "\n*********************************" << endl
<< "Welcome to the pi calculator!" << endl
<< "Programmer: K. Spry" << endl
<< "You set the number of divisions \nfor estimating the integral:
\n\tf(x)=4/(1+x^2)"
<< endl
<< "*********************************" << endl;
}                      //end printit
```

4.8 Το πρόβλημα N-Body

Κεντρικό ζητούμενο σε αυτό το πρόβλημα είναι να ευρεθούν οι θέσεις και οι κινήσεις των σωμάτων στο χώρο, όταν εξασκούνται βαρυτικές δυνάμεις μεταξύ των σωμάτων σύμφωνα με τους νευτώνειους νόμους της Φυσικής. Η ελκτική δύναμη που ασκείται μεταξύ δύο σωμάτων με μάζες m_a και m_b δίνεται από την ακόλουθη σχέση:

$$F = \frac{Gm_a m_b}{r^2},$$

όπου G είναι σταθερά και r είναι η απόσταση μεταξύ δύο σωμάτων. Επίσης, σύμφωνα με το δεύτερο νόμο του Νεύτωνα, όταν σε ένα σώμα ασκείται δύναμη F, το σώμα αποκτά επιτάχυνση γ σύμφωνα με τον τύπο:

$$F = m\gamma,$$

όπου m είναι η μάζα του σώματος. Για ένα μικρό διάστημα Δt, η επιτάχυνση γ μπορεί να γραφεί ως

$$\gamma = \frac{v(t + \Delta t) - v(t)}{\Delta t},$$

όπου $v(t + \Delta t)$ και $v(t)$ είναι οι ταχύτητα του σώματος τις χρονικές στιγμές $t + \Delta t$ και t. Η νέα ταχύτητα $v(t + \Delta t)$ δίνεται από τη σχέση:

$$v(t + \Delta t) = v(t) + \frac{F}{m}\Delta t.$$

Στο ίδιο χρονικό διάστημα Δt, η θέση του σώματος αλλάζει σύμφωνα με τον τύπο

$$x(t + \Delta t) - x(t) = v(t)\Delta t,$$

όπου $x(t)$ είναι η θέση του σώματος τη χρονική στιγμή t. Μόλις τα σώματα μετακινηθούν στις νέες τους θέσεις, οι δυνάμεις αλλάζουν και επομένως ο παραπάνω υπολογισμός πρέπει να επαναληφθεί.

4.8.1 Ακολουθιακός κώδικας για το πρόβλημα N-body

Συνολικά, ο υπολογισμός για το N-body πρόβλημα μπορεί να περιγραφεί ως ακολούθως:

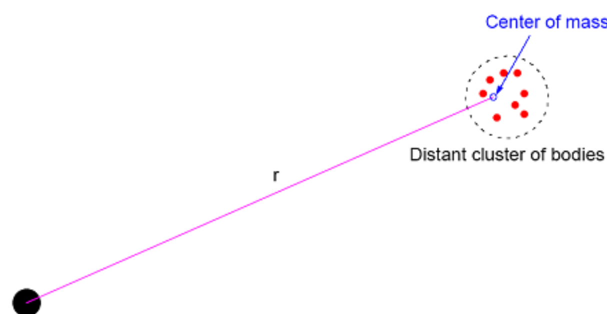
```
for (t = 0; t < tmax; t++)          /* for each time period */
    for (i = 0; i < N; i++) {       /* for each body */
        F = Force_routine(i);       /* compute force on ith body */
            v[i]new = v[i] + F * dt / m;
                                    /* compute new velocity */
            x[i]new = x[i] + v[i]new * dt;    /* and new position */
    }
for (i = 0; i < nmax; i++) {        /* for each body */
    x[i] = x[i]new;                 /* update velocity & position*/
    v[i] = v[i]new;
}
```

4.8.2 Παράλληλη υλοποίηση για το πρόβλημα N-body

Ο ακολουθιακός αλγόριθμος έχει πολυπλοκότητα $O(N^2)$ (για μία επανάληψη), καθώς κάθε ένα από τα N σώματα επηρεάζεται από τα υπόλοιπα $N-1$. Σε μια απλή στατική κατανομή έργων, κάθε μία από τις p συνολικά διεργασίες αναλαμβάνει τον υπολογισμό των ασκούμενων δυνάμεων σε n/p σώματα. Αυτή η κατανομή συνεπάγεται πυκνή κυκλοφορία μηνυμάτων (all-to-all broadcast), αφού κάθε διεργασία πρέπει να στέλνει σε όλες τις υπόλοιπες τις νέες συντεταγμένες των σωμάτων, μετά την μετατόπιση τους.

Εναλλακτικά, μπορούμε να ακολουθήσουμε την τεχνική master-slave, με τη master διεργασία να συντηρεί μία δεξαμενή έργων. Τα έργα σε αυτή την περίπτωση είναι ο υπολογισμός της ελκτικής δύναμης μεταξύ δύο σωμάτων. Με N σώματα, υπάρχουν συνολικά $N(N+1)/2$ τέτοια ζεύγη και επομένως $N(N+1)/2$ έργα προς εκτέλεση. Κάθε slave διεργασία αναλαμβάνει το επόμενο διαθέσιμο έργο από τη δεξαμενή της master διεργασίας και το αποτέλεσμα της επεξεργασίας (η ελκτική δύναμη μεταξύ των δύο σωμάτων) επιστρέφεται πίσω στη master διεργασία. Η master διεργασία προσθέτει τα αποτελέσματα των slave διεργασιών, για να υπολογίσει τη συνισταμένη δύναμη σε κάθε σώμα.

Η χρονική πολυπλοκότητα του ακολουθιακού αλγόριθμου μπορεί να μειωθεί, με τη παρατήρηση ότι μια ομάδα (cluster) από απομακρυσμένα σώματα μπορούν να προσεγγιστούν ως ένα μακρινό αντικείμενο, του οποίου η συνολική μάζα είναι τοποθετημένη στο κέντρο βάρους της ομάδας.



Σχήμα 4.15: Μια ομάδα από απομακρυσμένα σώματα.

Ο αλγόριθμος Barnes-Hut βασίζεται στη προηγούμενη παρατήρηση και έχει ως εξής:

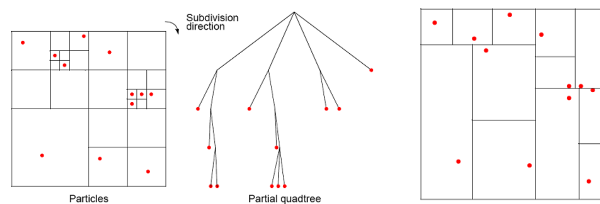
- Στην αρχή, όλος ο χώρος με τα N σώματα περιέχεται σε έναν μόνο κύβο.
- Στη συνέχεια, αυτός ο κόμβος διαιρείται σε οκτώ υποκύβους.
- Αν ένας υποκύβος δεν περιέχει σώματα, ο υποκύβος αυτός διαγράφεται και ο αλγόριθμος δεν ασχολείται με αυτόν περαιτέρω.
- Αν ένας υποκύβος περιέχει ένα σώμα, αυτός ο υποκύβος κρατείται.
- Αν ένας υποκύβος περιέχει περισσότερα του ενός σώματα, ο κύβος αυτός διαιρείται αναδρομικά μέχρι κάθε υποκύβος να περιέχει ένα μόνο σώμα.

Η προηγούμενη διαδικασία δημιουργεί ένα οκταδικό δέντρο (octtree), δηλαδή ένα δέντρο όπου κάθε κόμβος έχει το πολύ οκτώ παιδιά. Κάθε κόμβος του δέντρου αντιστοιχεί σε έναν υποκύβο και τα φύλλα αυτού του δέντρου αντιστοιχούν στους υποκύβους που περιέχουν ένα μόνο σώμα. Μετά τη κατασκευή του δέντρου, σε κάθε κόμβο αποθηκεύεται η συνολική μάζα και το κέντρο βάρους του υποκύβου που αντιστοιχεί στον κόμβο. Η δύναμη σε κάθε σώμα μπορεί να προσδιορισθεί με τη διάσχιση του δέντρου με αρχή τη ρίζα, μέχρι να συναντήσουμε έναν κόμβο όπου η παραπάνω προσέγγιση μπορεί να εφαρμοσθεί. Συγκεκριμένα, το κριτήριο για την εφαρμογή της προσέγγισης είναι η απόσταση r του υπό εξέταση σώματος από το κέντρο βάρους του cluster να ικανοποιεί την ακόλουθη ανισότητα:

$$r \geq \frac{d}{\theta},$$

όπου d είναι η διάσταση του υποκύβου και θ είναι μία σταθερά, συνήθως μικρότερη της μονάδας.

Τόσο η κατασκευή του δέντρου, όσο και ο υπολογισμός με βάση το δέντρο, έχει πολυπλοκότητα $O(N \log N)$. Με τη μετακίνηση των σωμάτων, το δέντρο θα πρέπει πάλι να κατασκευαστεί. Η παράλληλη υλοποίηση του αλγόριθμου Barnes-Hut παρουσιάζει δυσκολίες. Το οκταδικό δέντρο έχει μη κανονική μορφή και η κατανεμημένη υλοποίηση του δεν είναι εύκολη και συνήθως δημιουργεί προβλήματα ανισοκατανομής φόρτου εργασίας στις διεργασίες. Επίσης, η διάτρεξη του κατανεμημένου δέντρου, για τον υπολογισμό των δυνάμεων που ασκούνται στα σώματα, έχει ως αποτέλεσμα τη συχνή ανταλλαγή μηνυμάτων μεταξύ των διεργασιών.



Σχήμα 4.16:

Στο σχήμα 4.16, βλέπουμε ένα παράδειγμα αναδρομικής διαίρεσης του δισδιάστατου χώρου, μέχρι κάθε τετράγωνο να περιέχει ένα μόνο σώμα. Εναλλακτικά, μπορούμε να διαιρέσουμε το χώρο κατά τέτοιο τρόπο ώστε, σε κάθε επίπεδο της δενδρικής δομής τα υποδέντρα, να περιέχουν το ίδιο πλήθος σωμάτων. Η τεχνική αυτή είναι γνωστή ως *Αναδρομική Ορθογώνια Διχοτόμηση.*

Για 2-διάστατο χώρο, η διχοτόμηση αυτή επιτυγχάνεται ως εξής:

- Πρώτα, βρίσκεται η κάθετη γραμμή η οποία διαιρεί την περιοχή σε δύο περιοχές με το ίδιο πλήθος σωμάτων.
- Για κάθε μία από αυτές τις περιοχές, προσδιορίζεται η οριζόντια γραμμή που διαιρεί την περιοχή σε δύο νέες περιοχές με το ίδιο πλήθος σωμάτων.

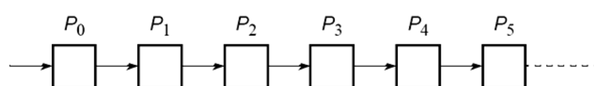
- Αυτή η διαδικασία επαναλαμβάνεται μέχρι να προκύψουν περιοχές που περιέχουν ένα μόνο σώμα.

Αυτός ο τρόπος διχοτόμησης του χώρου βοηθά στην εξισορρόπηση φορτιού (load balancing) σε μια παράλληλη υλοποίηση της μεθόδου.

Κεφάλαιο 5

Υπολογισμοί με την τεχνική της σωλήνωσης

Η τεχνική της σωλήνωσης (pipeline) εφαρμόζεται σε ένα μεγάλο εύρος προβλημάτων, τα οποία είναι εν μέρει ακολουθιακά από τη φύση τους, δηλαδή πρέπει να εκτελεστεί μία ακολουθία βημάτων. Πιο συγκεκριμένα, το πρόβλημα διαιρείται σε μια σειρά από έργα, τα οποία θα πρέπει να ολοκληρωθούν το ένα μετά το άλλο. Κάθε έργο εκτελείται από μία δαφορετική διεργασία/επεξεργαστή:



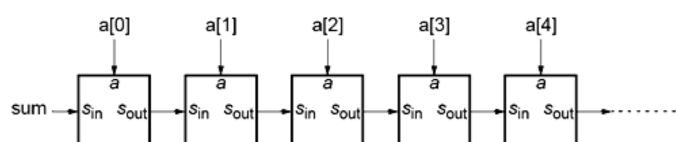
Σχήμα 5.1: Διεργασίες σε σωλήνωση.

Για παράδειγμα, ο υπολογισμός του αθροίσματος των στοιχείων ενός πίνακα μπορεί να υλοποιηθεί με τη βοήθεια της τεχνικής της σωλήνωσης:

```
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

Ο βρόχος μπορεί να «ξεδιπλωθεί» ως εξής:

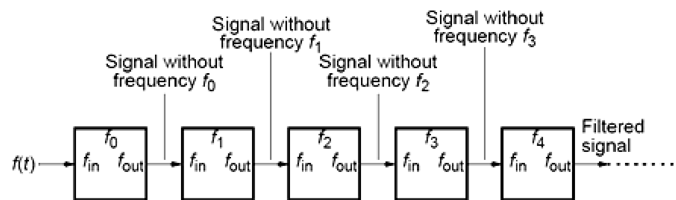
```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
....
```



Σχήμα 5.2: Υπολογισμός αθροίσματος με σωλήνωση.

Ο υπολογισμός αυτός μπορεί εύκολα να υλοποιηθεί με τη τεχνική της σωλήνωσης. Στη λύση αυτή, κάθε στάδιο αντιστοιχεί σε μία ξεχωριστή εντολή του ξεδιπλωμένου βρόχου. Κάθε στάδιο δέχεται ως είσοδο το συσσωρευμένο άθροισμα από τα προηγούμενα στάδια (είσοδος s_{in}) και το αντίστοιχο στοιχείο του πίνακα a, και παράγει το νέο συσσωρευμένο άθροισμα στην έξοδο s_{out}. Δηλαδή, το στάδιο i εκτελεί τον υπολογισμό $s_{out} = s_{in} + a[i]$.

Εκτός από απλές εντολές, μία ακολουθία από συναρτήσεις μπορούν να εκτελεστούν με την τεχνική της σωλήνωσης. Ένα φίλτρο συχνοτήτων είναι ένα τέτοιο παράδειγμα. Στο συγκεκριμένο παράδειγμα, ο στόχος είναι να αφαιρεθούν συγκεκριμένες συχνότητες (f_0, f_1, f_2, f_3, κτλ.) από ένα ψηφιακό σήμα $f(t)$. Το σήμα εισέρχεται στη σωλήνωση από αριστερά. Σε κάθε στάδιο εφαρμόζεται η συνάρτηση που αφαιρεί μία συχνότητα από το ψηφιακό σήμα.



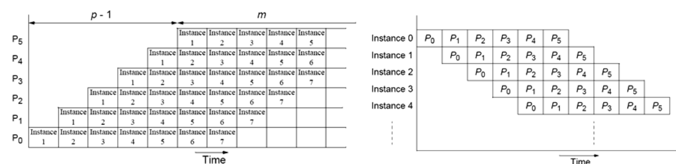
Σχήμα 5.3: Φίλτρο συχνοτήτων με σωλήνωση.

Πότε η τεχνική της σωλήνωσης έχει επιτυχία. Υποθέτοντας ότι το υπό εξέταση πρόβλημα μπορεί να διαιρεθεί σε μια ακολουθία από έργα, η τεχνική της σωλήνωσης μπορεί να δώσει χαμηλότερους χρόνους εκτέλεσης σε τρεις περιπτώσεις:

1. Αν υπάρχουν περισσότερα από ένα στιγμιότυπα του ίδιου προβλήματος που πρέπει να επιλυθούν. (Type 1 pipeline.)

2. Αν πρέπει να γίνει επεξεργασία μιας ακολουθίας δεδομένων και η επεξεργασία κάθε δεδομένου είναι σύνθετη, απαιτώντας μια σειρά από λειτουργίες. (Type 2 pipeline.)

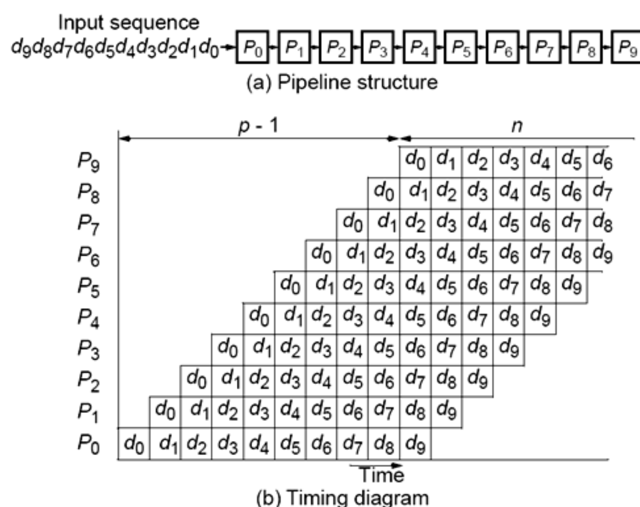
3. Αν κάθε στάδιο i της σωλήνωσης μπορεί να δώσει γρήγορα τμήμα του αποτελέσματος του στο επόμενο στάδιο, πριν να ολοκληρωθεί πλήρως η επεξεργασία στο στάδιο i. Το επόμενο στάδιο μπορεί να αρχίσει την επεξεργασία του, βασιζόμενο μόνο στην αρχική πληροφορία που στέλνει το στάδιο i. Στην πορεία, λαμβάνει και το υπόλοιπο αποτέλεσμα του σταδίου i. (Type 3 pipeline.)



Σχήμα 5.4: Type 1 pipeline - Χρονοδιάγραμμα.

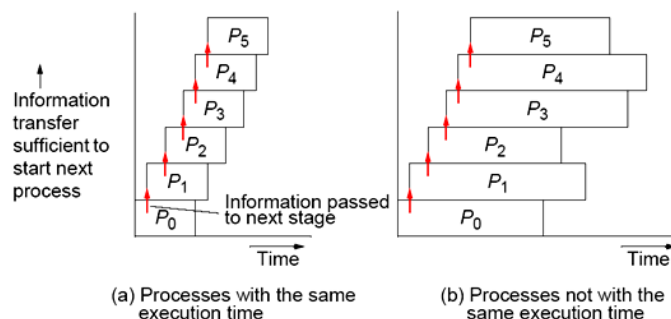
Ο τύπος 1 της τεχνικής της σωλήνωσης χρησιμοποιείται ευρέως για το σχεδιασμό των επεξεργαστών στο επίπεδο του υλικού. Υπάρχουν πολλαπλά στιγμιότυπα του ίδιου προβλήματος και για την επίλυση ενός στιγμιότυπου απαιτούνται p σταδια επεξεργασίας. Κάθε στάδιο εκτελείται από μία διαφορετική διεργασία. Όταν η διεργασία ολοκληρώσει την επεξεργασία της για ένα στιγμιότυπο του προβλήματος, λαμβάνει από την προηγούμενη διεργασία τα ενδιάμεσα

αποτελέσματα για το επόμενο στιγμιότυπο και στη συνέχεια εκτελεί το στάδιο που έχει αναλάβει για το νέο στιγμιότυπο. Ο συνολικός χρόνος για την επίλυση των m στιγμιότυπων θα είναι $m + p - 1$ βήματα.



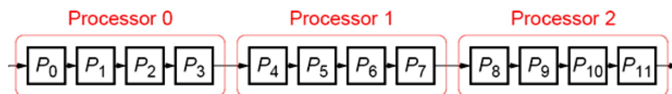
Σχήμα 5.5: Type 2 pipeline - Χρονοδιάγραμμα.

Στον τύπο 2 της τεχνικής της σωλήνωσης, υπάρχει μια ακολουθία δεδομένων, στα οποία πρέπει να γίνει επεξεργασία διαδοχικά. Σε κάθε δεδομένο αυτής της ακολουθίας, υπάρχουν πολλαπλά στάδια επεξεργασίας που θα πρέπει να εκτελεστούν και τα στάδια αυτά τα αναλαμβάνουν ισάριθμες διεργασίες. Ο συνολικός χρόνος για την επεξεργασία n δεδομένων είναι $n + p - 1$ βήματα.



Σχήμα 5.6: Type 3 pipeline - Χρονοδιάγραμμα.

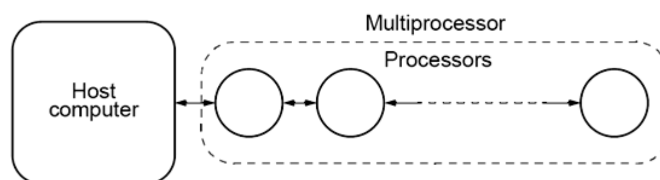
Ο τύπος 3 της τεχνικής της σωλήνωσης είναι ο τύπος που χρησιμοποιείται στα παράλληλα προγράμματα. Πριν τελειώσει η επεξεργασία της, κάθε διεργασία i περνάει πληροφορία στην επόμενη διεργασία $i + 1$. Η υπόλοιπη πληροφορία περνάει σταδιακά στη διεργασία $i + 1$, καθώς παράγεται από τη διεργασία i. Αν το πλήθος των σταδίων σε μια σωλήνωση είναι μεγαλύτερος από το πλήθος των διαθέσιμων διεργασιών/επεξεργαστών, κάθε διεργασία μπορεί να αναλάβει μια ομάδα διαδοχικών σταδίων.



Σχήμα 5.7: Ομάδες σταδίων, διαμοιρασμένες σε διαφορετικές διεργασίες.

5.1 Υπολογιστικές πλατφόρμες κατάλληλες για την τεχνική της σωλήνωσης

Το ιδανικό διασυνδετικό δίκτυο για τη τεχνική της σωλήνωσης είναι η γραμμή ή ο δακτύλιος. Τέτοιου είδους δίκτυα μπορούν εύκολα να προσομοιωθούν από mesh δίκτυα και τα δίκτυα υπερκύβου. Παρά τις περιορισμένες δυνατότητες επικοινωνίας που έχει μια γραμμική διάταξη, πολλές εφαρμογές μπορούν να υλοποιηθούν σε τέτοιες διατάξεις με χαμηλό κόστος. Σε μία αρχιτεκτονική cluster, η τεχνική της σωλήνωσης μπορεί να εφαρμοσθεί επιτυχώς μόνο αν το διασυνδετικό δίκτυο των υπολογιστών επιτρέπει ταυτόχρονες μεταφορές μεταξύ επεξεργαστών.

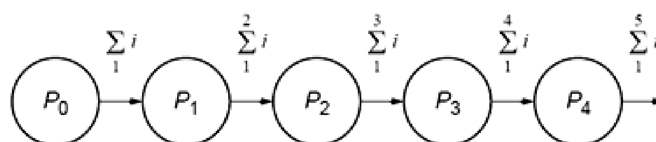


Σχήμα 5.8: Διασυνδετικό δίκτυο.

5.2 Παραδείγματα λύσεων με βάση την τεχνική της σωλήνωσης

5.2.1 Άθροιση αριθμών - Σωλήνωση Τύπου 1

Γίνεται εφαρμογή της τεχνικής σωλήνωσης τύπου 1.



Σχήμα 5.9: Άθροιση αριθμών.

Ο κώδικας που εκτελείται σε όλες τις διεργασίες P_i (i), εκτός από τις διεργασίες P_0 και P_1 είναι ο ακόλουθος:

```
recv(&accumulation, i-1);
accumulation = accumulation + number;
send(&accumulation, i+1);
```

όπου accumulation είναι το μερικό άθροισμα, όπως έχει σχηματιστεί από τις προηγούμενες διεργασίες, και number είναι το στοιχείο $a[i]$ που θα προστεθεί στο μερικό άθροισμα accumulation. Η διεργασία P_0 εκτελεί τον εξής κώδικα:

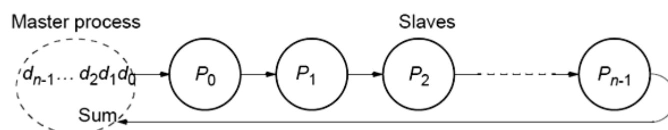
```
send(&number, 1)
```

Επίσης η διεργασία P_{n-1} εκτελεί τις εξής εντολές:

```
recv(&number, n-2);
accumulation = accumulation + number;
```

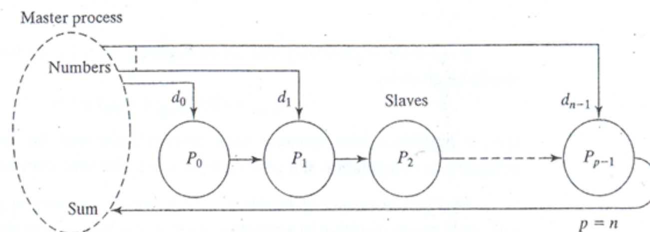
Συνολικά το SPMD πρόγραμμα θα έχει ως εξής:

```
if (process > 0) {
    recv(&accumulation, i-1);
    accumulation = accumulation + number;
}
if (process < n-1)
    send(&accumulation, i+1);
```



Σχήμα 5.10:

Υπάρχουν δύο παραλλαγές στη βασική τεχνική, όσον αφορά τον τρόπο με τον οποίο εισάγονται οι αριθμοί στη διάταξη σωλήνωσης. Στην πρώτη περίπτωση, η master και οι slave διεργασίες είναι σε διάταξη δακτυλίου και τα δεδομένα εισόδου εισάγονται από τη master διεργασία μέσω της διαδικασίας P_0 σε όλες τις υπόλοιπες διεργασίες. Εναλλακτικά, η master διεργασία δίνει κατευθείαν τα δεδομένα στις slave διεργασίες, όταν αυτές τα χρειάζονται, δηλαδή στον πρώτο κύκλο δίνει το στοιχείο d_0 στην P_0, στο δεύτερο κύκλο δίνει το στοιχείο d_1 στην P_1 κοκ.



Σχήμα 5.11:

Στη δεύτερη περίπτωση, ο συνολικός χρόνος για τον υπολογισμό του αθροίσματος n στοιχείων θα είναι

$$t_{total} = (t_{comp} + t_{comm})n,$$

όπου $t_{comp} = 1$ είναι το κόστος της άθροισης σε κάθε διεργασία και $t_{comm} = (t_{startup} + t_{data})$ είναι το κόστος επικοινωνίας μεταξύ γειτονικών διεργασιών. Αν υπάρχουν m στιγμιότυπα του

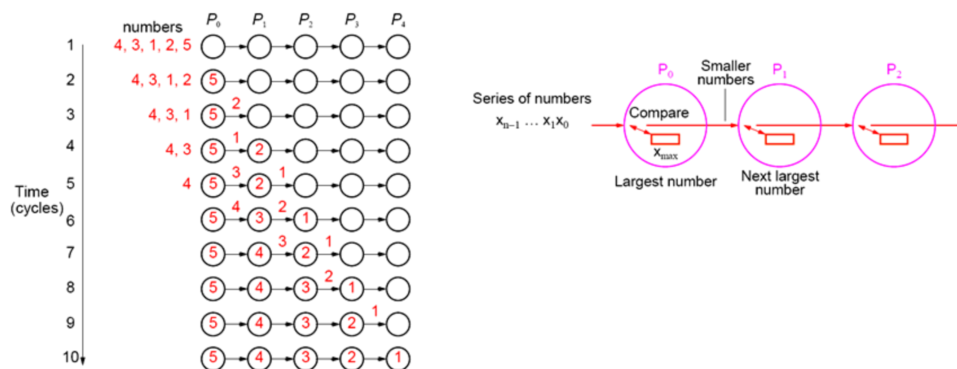
προβλήματος της άθροισης των n αριθμών που πρέπει να επιλυθούν, η συνολική πολυπλοκότητα του θα είναι

$$t_{total} = (t_{comp} + t_{comm})(m + n - 1).$$

Ο μέσος χρόνος για την ολοκλήρωση ενός στιγμιότυπου θα είναι $t_a = t_{total}/m$. Για μεγάλη τιμή του m, ο χρόνος αυτός θα είναι περίπου ίσος με $t_{comp} + t_{comm}$.

5.2.2 Ταξινόμηση - Σωλήνωση Τύπου 2

Ο αλγόριθμος ταξινόμησης παρεμβολής (insertion sort) είναι ένας γνωστός αλγόριθμος ταξινόμησης. Μπορεί να υλοποιηθεί παράλληλα με τη βοήθεια της τεχνικής της σωλήνωσης. Συγκεκριμένα, η διεργασία P_0 δέχεται διαδοχικά τα στοιχεία που θα ταξινομηθούν, αποθηκεύει το μεγαλύτερο αριθμό που έχει δει μέχρι τώρα και περνάει στην επόμενη διεργασία όλους τους υπόλοιπους αριθμούς. Αν ένα νέο στοιχείο ληφθεί, που είναι μεγαλύτερο από αυτό που έχει αποθηκευθεί, το ήδη αποθηκευμένο στοιχείο στέλνεται στην επόμενη διεργασία και το νέο στοιχείο παίρνει τη θέση του. Οι υπόλοιπες διεργασίες εκτελούν την ίδια διαδικασία, δηλαδή κρατούν πάντα το μεγαλύτερο μέχρι εκείνη τη στιγμή στοιχείο και περνούν τους υπόλοιπους αριθμούς στην επόμενη διεργασία.



Σχήμα 5.12:

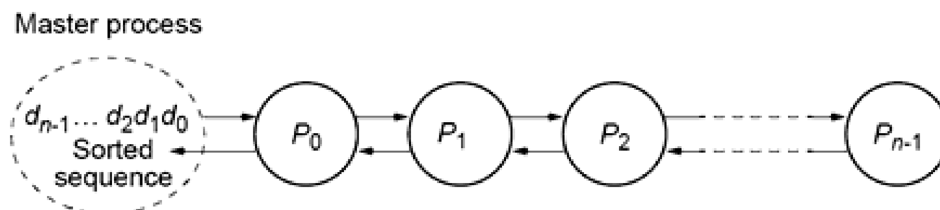
Ο βασικός αλγόριθμος για τη διεργασία P_i είναι:

```
right_procNum = n-i-1;
recv(&x, Pi-1);
for (j=0; j<right_procNum; j++) {
    recv(&number, i-1);
    if (number > x) {
        send(&x, i+1);
    x = number;
    } else send(&number, i+1);
}
```

/* τα στοιχεία του ταξινομημένου πίνακα ολισθαίνουν προς τα «αριστερά» και συλλέγονται από τη master διεργασία */

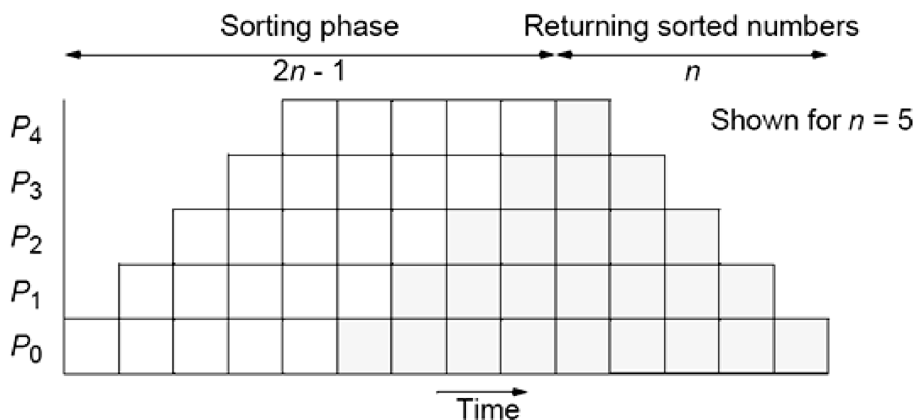
```
send(&x, i-1)
for (j=0; j<right_procNum; j++) {
    recv(&number, i+1);
    send(&number, i-1);
}
```

Κάθε διεργασία γνωρίζει ότι θα λάβει $n-i$ αριθμούς. Επίσης, γνωρίζει ότι θα περάσει $n-i-1$ στοιχεία στην επόμενη διεργασία, αφού ένα στοιχείο θα αποθηκευθεί στη διεργασία.



Σχήμα 5.13:

Συνολικά, τα στοιχεία κινούνται προς τα δεξιά κατά τη φάση της ταξινόμησης και προς τα αριστερά όταν συλλέγεται ο ταξινομημένος πίνακας από τη master διεργασία.



Σχήμα 5.14: Χρονοδιάγραμμα της ταξινόμησης παρεμβολής.

Ο χρόνος εκτέλεσης της φάσης ταξινόμησης θα είναι

$$(2n-1)\cdot(t_{comp}+t_{comm}),$$

ενώ ο χρόνος για τη συλλογή του ταξινομημένου πίνακα από τη master διεργασία είναι $n \cdot t_{comm}$.

5.2.3 Παραγωγή πρώτων αριθμών - Τύπος 2 σωλήνωσης

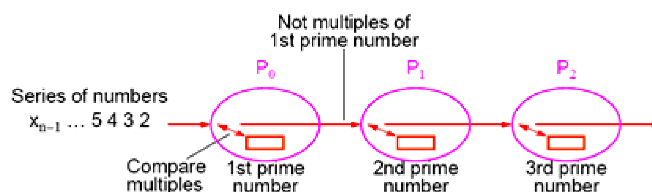
Με τον αλγόριθμο που είναι γνωστός ως Κόσκινο του Ερατοσθένη, βρίσκουμε όλους τους πρώτους αριθμούς που είναι μικρότεροι από έναν αριθμό n. Ο ακολουθιακός αλγόριθμος διαγράφει πρώτα όλα τα πολλαπλάσια του 2 που είναι μικρότερα του n. Στη συνέχεια, διαγράφει όλα τα πολλαπλάσια του μικρότερου των στοιχείων που έχουν απομείνει από τον πρώτο γύρο. Η διαδικασία αυτή επαναλαμβάνεται, μέχρι να απομείνουν μόνο οι πρώτοι αριθμοί μικρότεροι του n. Η διαδικασία αυτή μπορεί να υλοποιηθεί παράλληλα με τη βοήθεια της τεχνικής σωλήνωσης.

Ο κώδικας για την i-οστή διεργασία έχει ως εξής:

```
recv(&x, i-1);
for (i = 0; i < n; i++) {
```

```
    recv(&number, i-1);
    if (number % x) != 0 or number == terminator) send(&number, Pi+1);
    if (number == terminator) break;
}
```

Στο τέλος, η διεργασία i θα αποθηκεύσει τον i-στό μικρότερο πρώτο αριθμό. Επίσης, κάθε διεργασία δεν θα λάβει το ίδιο πλήθος δεδομένων και επιπλέον η ποσότητα αυτή δεν είναι γνωστή εκ των προτέρων. Για το λόγο αυτό, χρησιμοποιείται και ένα μήνυμα "terminator", που στέλνεται στο τέλος της ακολουθίας των αριθμών. Κάθε διαδικασία που λαμβάνει αυτό το μήνυμα, γνωρίζει ότι δεν θα λάβει άλλα δεδομένα και, αφού στείλει το μήνυμα στην επόμενη διεργασία, στη συνέχεια τερματίζει. Η ανάλυση πολυπλοκότητας είναι παρόμοια με αυτή του αλγόριθμου ταξινόμησης, με τη διαφορά ότι τώρα κάθε διεργασία εκτελεί λιγότερα βήματα από ότι η προηγούμενη διεργασία, επειδή δεν θα λάβει όλους τους αριθμούς που θα λάβει η προηγούμενη διεργασία.



Σχήμα 5.15: Σωλήνωση για την παραγωγή πρώτων αριθμών.

5.2.4 Επίλυση συστήματος γραμμικών εξισώσεων (άνω τριγωνική μορφή) - Τύπος 3 σωλήνωσης

Στόχος είναι η επίλυση ενός συστήματος γραμμικών εξισώσεων, που είναι σε άνω τριγωνική μορφή, όπου τα στοιχεία $a_{i,j}$ και b_i είναι σταθερές και x_i είναι οι άγνωστες μεταβλητές που πρέπει να προσδιορισθούν.

$$\begin{array}{ccccccccc} a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & \cdots & + & a_{n-2,2}x_2 & = & b_{n-1} \\ \vdots & & \vdots & & \iddots & & \vdots & \vdots & \vdots \\ a_{1,0}x_0 & + & a_{1,1}x_1 & & & & & = & b_1 \\ a_{0,0}x_0 & & & & & & & = & b_0 \end{array}$$

Για την επίλυση του γραμμικού συστήματος, χρησιμοποιούμε την προς τα πίσω αντικατάσταση (back substitution). Πρώτα, ο άγνωστος x_0 βρίσκεται κατευθείαν από την τελευταία εξίσωση, δηλαδή

$$x_0 = \frac{b_0}{a_{0,0}}.$$

Στη συνέχεια, γίνεται αντικατάσταση της τιμής του x_0 στην επόμενη εξίσωση και έτσι λαμβάνεται η μεταβλητή x_1, δηλαδή

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}.$$

Έπειτα, οι τιμές για τους αγνώστους x_1 και x_0 αντικαθίσταται στη επόμενη εξίσωση, για να λάβουμε τη τιμή για τη μεταβλητή x_2:

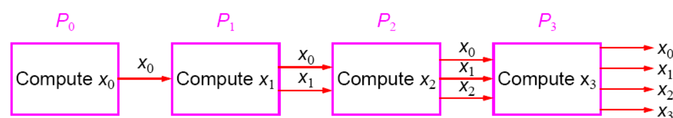
$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}.$$

Η διαδικασία αυτή επαναλαμβάνεται μέχρι να προσδιορισθούν οι τιμές για όλους τους αγνώστους.

Η επίλυση ενός γραμμικού άνω τριγωνικού συστήματος μπορεί να υλοποιηθεί παράλληλα με την τεχνική της σωλήνωσης. Το πρώτο στάδιο της σωλήνωσης (διεργασία P_0) υπολογίζει τον άγνωστο x_0 και περνάει την τιμή x_0 στο δεύτερο στάδιο (διεργασία P_1), το οποίο υπολογίζει την τιμή της μεταβλητής x_1 από τη x_0 και περνάει και τις δύο τιμές x_0 και x_1 στο επόμενο στάδιο (διεργασία P_2), το οποίο υπολογίζει την τιμή της x_2 από τις x_0 και x_1 κοκ.

Γενικά, η i-οστή διεργασία ($0 < i < n$) λαμβάνει τις τιμές $x_0, x_1, x_2, \ldots, x_{i-1}$ και υπολογίζει το x_i από την εξίσωση:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}.$$



Σχήμα 5.16: Επίλυση άνω τριγωνικού γραμμικού συστήματος με σωλήνωση.

Ο ακολουθιακός κώδικας για τη συγκεκριμένη μέθοδο έχει ως εξής:

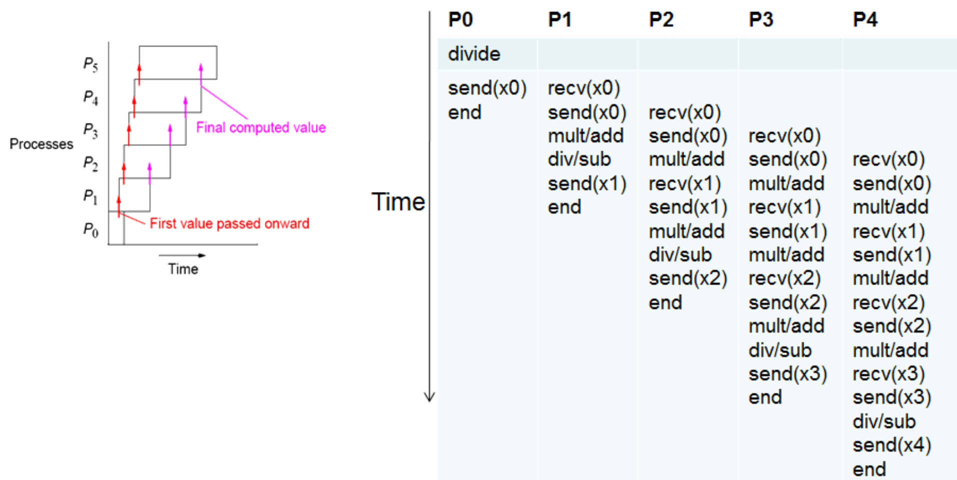
```
x[0] = b[0]/a[0][0];              /* computed separately */
for (i = 1; i < n; i++) {          /*for remaining unknowns*/
    sum = 0;
    for (j = 0; j < i; j++)
        sum = sum + a[i][j]*x[j];
    x[i] = (b[i] - sum)/a[i][i];
}
```

Στη παράλληλη υλοποίηση της ίδιας μεθόδου, ο κώδικας της διεργασίας P_i έχει ως εξής:

```
sum=0;
for (j = 0; j < i; j++) {
    recv(&x[j], i-1);
    send(&x[j], i+1);
    sum = sum + a[i][j]*x[j];
}
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], i+1);
```

Η διεργασία i λαμβάνει τις τιμές των στοιχείων $x[0], x[1], x[2], \ldots, x[i-1]$ διαδοχικά. Κάθε φορά που λαμβάνει μια τιμή $x[j]$, υπολογίζει το γινόμενο με την τιμή $a[i][j]$ και συσσωρεύει το αποτέλεσμα στη μεταβλητή sum. Μετά τη λήψη των τιμών όλων των μεταβλητών, μπορεί να υπολογίσει την τιμή της μεταβλητής $x[i]$.

Παρατηρείστε ότι κάθε διεργασία προωθεί αποτελέσματα στις επόμενες διεργασίες, πριν να ολοκληρωθεί πλήρως η τοπική επεξεργασία. Η πρώτη διεργασία P_0 εκτελεί μία διαίρεση και ένα send. Η i-οστή διεργασία ($0 < i < p-1$) εκτελεί i recv(), i send(), i πολλαπλασιασμούς/προσθέσεις, μία διαίρεση/αφαίρεση και ένα τελικό send, συνολικά $(2i + 1)$ βήματα επικοινωνίας και $(2i + 2)$ βήματα υπολογισμού. Η διαδικασία P_{p-1} εκτελεί $p - 1$ βήματα επικοινωνίας και $2p - 1$ βήματα υπολογισμού. Στο δεξιό σχήμα αναπαρίσταται η εκτέλεση των εντολών στις διάφορες διεργασίες. Γίνεται η υπόθεση ότι όλες οι λειτουργίες (υπολογισμού και επικοινωνίας) απαιτούν τον ίδιο



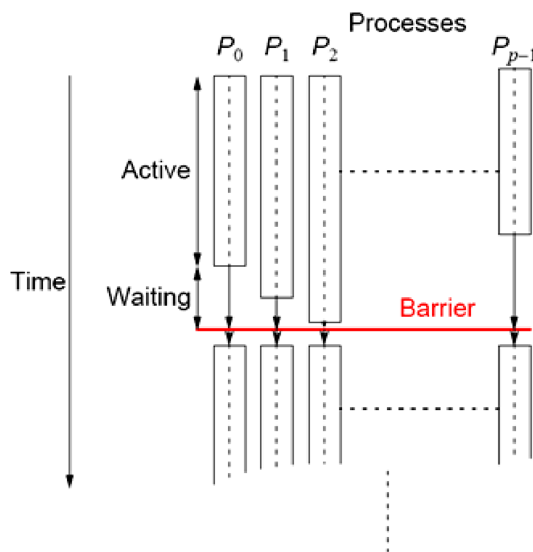
Σχήμα 5.17: Χρονοδιάγραμμα για την επίλυση άνω τριγωνικού γραμμικού συστήματος με σωλήνωση.

χρόνο εκτέλεσης. Ο συνολικός χρόνος εκτέλεσης δίνεται από το χρόνο εκτέλεσης της τελευταίας διεργασίας συν το χρόνο εκτέλεσης $p-1$ send συν το χρόνο μίας διαίρεσης.

Κεφάλαιο 6

Συγχρονισμένοι υπολογισμοί

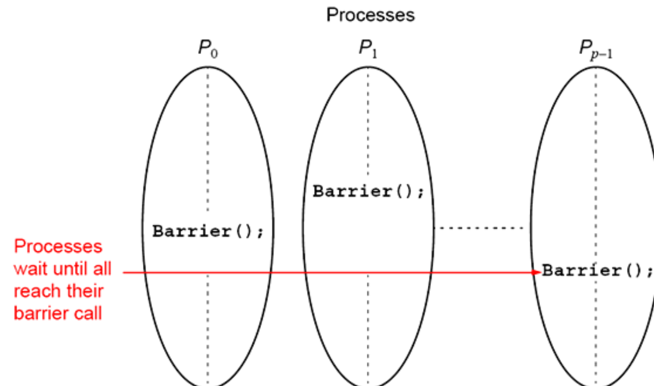
Σε πολλά προβλήματα, οι διεργασίες του παράλληλου προγράμματος, αφού εκτελέσουν μια σειρά από υπολογισμούς, θα πρέπει να περιμένουν η μία την άλλη, πριν προχωρήσουν περαιτέρω τους υπολογισμούς τους. Αυτή η αναμονή συνήθως επιβάλλεται από το γεγονός ότι κάθε διεργασία χρειάζεται τα αποτελέσματα άλλων διεργασιών. Σε μια πλήρως σύγχρονη εφαρμογή, όλες οι διεργασίες συγχρονίζονται ανά τακτά χρονικά διαστήματα. Ο βασικός μηχανισμός με τον οποίο οι διεργασίες συγχρονίζονται είναι ο μηχανισμός Barrier (φράγμα). Ο μηχανισμός εισάγεται στο σημείο εκτέλεσης στο οποίο κάθε διεργασία πρέπει να περιμένει. Οι διεργασίες μπορούν να συνεχίσουν την εκτέλεση των εντολών τους πέρα από αυτό το σημείο, όταν όλες οι διεργασίες έχουν φτάσει στο σημείο αυτό κατά την εκτέλεσή τους (ή, σε μερικές υλοποιήσεις, όταν ένα συγκεκριμένος πλήθος διεργασιών έχει φτάσει σε αυτό το σημείο).



Σχήμα 6.1: Παράδειγμα συγχρονισμού διεργασιών με φράγμα.

Στο παράδειγμα του Σχήματος 6.1, οι διαδικασίες φτάνουν στο φράγμα σε διαφορετικές χρονικές στιγμές. Συγκεκριμένα, η P_2 φτάνει αργότερα στο φράγμα σε σχέση με τις υπόλοιπες διεργασίες και έτσι όλες οι υπόλοιπες διεργασίες περιμένουν την P_2 να φτάσει στο φράγμα. Από τη στιγμή που όλες οι διεργασίες φτάσουν στο φράγμα, όλες οι διεργασίες μπορούν να συνεχίσουν την εκτέλεση των εντολών τους. Ο μηχανισμός των φραγμάτων βρίσκει εφαρμογή τόσο στο προγραμματισμό συστημάτων διαμοιραζόμενης μνήμης όσο και στα συστήματα κατανεμημένης μνήμης. Προς το παρόν, θα δοθούν λεπτομέρειες για τα φράγματα σε συστήματα κατανεμημέ-

νης μνήμης. Αργότερα, θα περιγραφεί αυτός ο μηχανισμός και στα συστήματα διαμοιραζόμενης μνήμης.

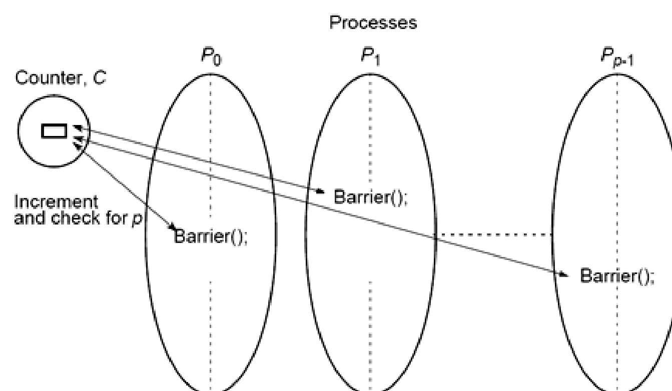


Σχήμα 6.2: Παράδειγμα συγχρονισμού διεργασιών με φράγμα.

Στα συστήματα κατανεμημένης μνήμης, τα φράγματα παρέχονται μέσω συναρτήσεων βιβλιοθηκών. Το MPI προσφέρει το μηχανισμό φράγματος μέσω της συνάρτησης MPI_Barrier(). Το μοναδικό όρισμα της συνάρτησης είναι ο communicator, στα όρια του οποίου λαμβάνει χώρα ο συγχρονισμός. Κάθε διεργασία που ανήκει στο group του communicator εκτελεί την MPI_Barrier() και αναστέλλει την εκτέλεσή της, μέχρι να εκτελέσουν τη συγκεκριμένη εντολή όλες οι διεργασίες του group. Το MPI δεν προσδιορίζει τον τρόπο υλοποίησης του MPI_Barrier(). Υπάρχουν πολλοί τρόποι υλοποίησης του βασικού μηχανισμού στα συστήματα κατανεμημένης μνήμης.

6.1 Υλοποίηση φράγματος

Η πρώτη υλοποίηση είναι συγκεντρωτική (centralized) με τη βοήθεια μετρητή (γραμμικό φράγμα). Συγκεκριμένα, υπάρχει ένας μετρητής που μετράει το πλήθος των διεργασιών που έχουν φτάσει το φράγμα. Κάθε διεργασία που φτάνει στο φράγμα αυξάνει το μετρητή και στη συνέχεια ελέγχει αν ο μετρητής είναι μικρότερος από το συνολικό πλήθος p των διεργασιών. Σε αυτή την περίπτωση, αναστέλλει την εκτέλεσή της. Όταν ο μετρητής γίνει ίσος με p, όλες οι διεργασίες που έχουν ακινητοποιηθεί στο φράγμα, ενεργοποιούνται και συνεχίζουν την εκτέλεσή τους.



Σχήμα 6.3: Υλοποίηση φράγματος.

Μια καλή υλοποίηση για το μηχανισμό του φράγματος θα πρέπει να παίρνει υπόψη ότι μια διεργασία μπορεί να χρησιμοποιήσει το φράγμα περισσότερες από μία φορά. Είναι επίσης πιθανόν μια διεργασία να φτάσει στο φράγμα και δεύτερη φορά προτού φύγουν οι προηγούμενες διεργασίες από το φράγμα από την πρώτη φορά. Οι δύο παραπάνω περιπτώσεις μπορούν εύκολα να αντιμετωπιστούν με τη χρήση δύο φάσεων στην υλοποίηση του γραμμικού φράγματος:

- **Φάση άφιξης**: Μια διεργασία εισέρχεται στη φάση άφιξης και δεν εγκαταλείπει αυτή τη φάση μέχρι όλες οι διεργασίες να εισέρθουν σε αυτή τη φάση.

- **Φάση αναχώρησης**: Οι διεργασίες μεταβαίνουν στη φάση αναχώρησης και εγκαταλείπουν το φράγμα.

Παράδειγμα:
Master process:

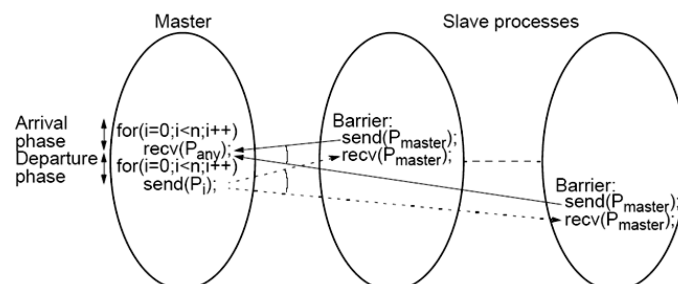
```
for (i = 0; i < n; i++)       /*count slaves as they reach barrier*/
    recv(Pany);
for (i = 0; i < n; i++)      /* release slaves */
    send(Pi);
```

Slave processes:

```
send(Pmaster);
recv(Pmaster);
```

6.1.1 Υλοποίηση του γραμμικού φράγματος με master-slave τεχνική

Η master διεργασία ελέγχει τη μεταβλητή *i*, η οποία είναι ο μετρητής που μετράει πόσες διεργασίες έχουν φτάσει στο φράγμα. Οι slave διεργασίες εισέρχονται με τυχαία σειρά στη φάση άφιξης, ενώ εγκαταλείπουν τη φάση αναχώρησης με συγκεκριμένη σειρά, η οποία καθορίζεται από τη ακολουθία εκτέλεσης των εντολών send από τη master διεργασία. Σημειώνεται ότι οι εντολές recv είναι blocking, δηλαδή η διεργασία που την εκτελεί περιμένει μέχρι να ληφθούν τα δεδομένα τα οποία αναμένει η recv.



Σχήμα 6.4: Υλοποίηση του γραμμικού φράγματος με master-slave τεχνική.

6.1.2 Υλοποίηση φράγματος με δέντρο

Το γραμμικό φράγμα έχει πολυπλοκότητα $O(p)$, όπου p είναι το πλήθος των διεργασιών που συγχρονίζονται. Πιο αποδοτική υλοποίηση $O(\log p)$ βημάτων μπορούμε να πετύχουμε με υλοποίηση με τη βοήθεια δέντρου. Ας υποθέσουμε ότι έχουμε 8 διεργασίες, $P_0, P_1, \ldots P_7$:

Σε πρώτο στάδιο, οι διεργασίες εκτελούν τις ακόλουθες εντολές:

- Η P_1 στέλνει μήνυμα στην P_0 (όταν η P_1 φτάσει στο φράγμα).
- Η P_3 στέλνει μήνυμα στην P_2 (όταν η P_3 φτάσει στο φράγμα).
- Η P_5 στέλνει μήνυμα στην P_4 (όταν η P_5 φτάσει στο φράγμα).
- Η P_7 στέλνει μήνυμα στην P_6 (όταν η P_7 φτάσει στο φράγμα).

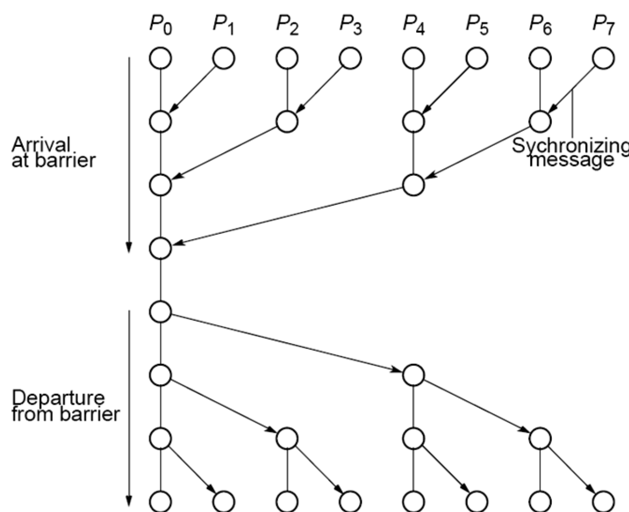
Σε δεύτερο στάδιο, οι διεργασίες εκτελούν τις ακόλουθες εντολές:

- Η P_2 στέλνει μήνυμα στην P_0. (Οι P_2 και P_3 έχουν φτάσει στο φράγμα.)
- Η P_6 στέλνει μήνυμα στην P_4. (Οι P_6 και P_7 έχουν φτάσει στο φράγμα.)

Σε τρίτο στάδιο, οι διεργασίες εκτελούν τις ακόλουθες εντολές:

- Η P_4 στέλνει μήνυμα στην P_0. (Οι P_4, P_5, P_6 και P_7 έχουν φτάσει στο φράγμα.)

Τέλος, η P_0 τερματίζει τη φάση άφιξης (όταν η P_0 φτάσει στο φράγμα και λάβει μήνυμα από την P_4). Στη φάση αναχώρησης, η παραπάνω διαδικασία αντιστρέφεται, με τη διεργασία P_0 να στέλνει μήνυμα «αναχώρησης» σε όλες τις άλλες διεργασίες.

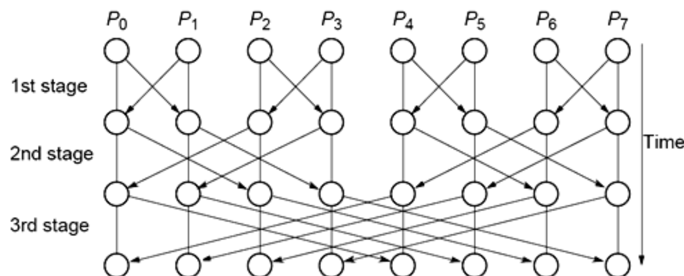


Σχήμα 6.5: Υλοποίηση φράγματος με δέντρο.

6.1.3 Υλοποίηση φράγματος με πεταλούδα

Σε αυτή την υλοποίηση, ζεύγη διεργασιών συγχρονίζονται σε κάθε στάδιο. Κάθε τόξο στο Σχήμα 6.6 είναι επικοινωνία μεταξύ διεργασιών με send()/recv(). Μετά την ολοκλήρωση των τριών σταδίων, κάθε διεργασία γνωρίζει ότι όλες οι άλλες διεργασίας έχουν φτάσει επίσης στο φράγμα. Στη φάση s, η διεργασία i συγχρονίζεται με τη διεργασία $i + 2s - 1$, αν p είναι δύναμη του 2. Αν το πλήθος p των διεργασιών δεν είναι δύναμη του 2, η επικοινωνία είναι μεταξύ της διεργασίας i και της διεργασίας $(i + 2s - 1) \bmod p$. Η χρονική πολυπλοκότητα της υλοποίησης είναι $O(\log p)$.

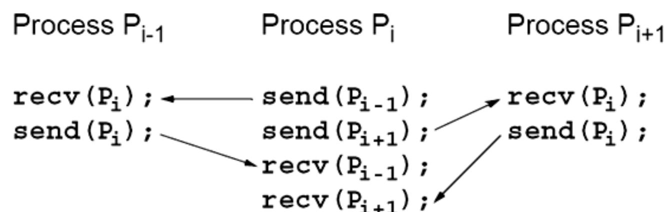
1st stage $P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$
2nd stage $P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
3rd stage $P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$



Σχήμα 6.6: Υλοποίηση φράγματος με πεταλούδα.

6.2 Τοπικός Συγχρονισμός

Σε κάποια προβλήματα, οι διεργασίες πρέπει να συγχρονίζονται με ορισμένες μόνο διεργασίες και όχι με άλλες. Τέτοια παραδείγματα προκύπτουν όταν οι διεργασίες οργανώνονται σε πλέγμα ή σε διάταξη σωλήνωσης. Σε αυτές τις περιπτώσεις, χρειάζεται να συγχρονιστούν μόνο με τους γείτονές τους. Στο παρακάτω παράδειγμα, η διεργασία P_i πρέπει να συγχρονιστεί και ανταλλάξει δεδομένα με τη διεργασία P_{i-1} και τη διεργασία P_{i+1} πριν συνεχίσει:



Σχήμα 6.7: Τοπικός Συγχρονισμός.

Στην ουσία, δεν πρόκειται για τέλειο συγχρονισμό μεταξύ των τριών διεργασιών, επειδή η διεργασία P_{i-1} συγχρονίζεται μόνο με την P_i και συνεχίζει μόλις η P_i το επιτρέψει. Παρόμοια, η διεργασία P_{i+1} συγχρονίζεται μόνο με την P_i.

6.3 Αδιέξοδο (deadlock)

Στις υλοποιήσεις φράγματος με δέντρο, πεταλούδα ή στον τοπικό συγχρονισμό, όταν ένα ζεύγος διεργασιών επικοινωνεί με send()/recv(), υπάρχει πιθανότητα να προκύψει αδιέξοδο που θα ακινητοποιήσει τις εμπλεκόμενες διεργασίες. Το αδιέξοδο θα συμβεί εάν και οι δύο διεργασίες εκτελούν την εντολή send, χρησιμοποιώντας σύγχρονες ρουτίνες πρώτα (ή blocking ρουτίνες χωρίς επαρκή αποθηκευτικό χώρο (buffering». Αυτό συμβαίνει επειδή καμία διεργασία δεν επιστρέφει από την εντολή send, αφού θα περιμένει για το αντίστοιχο recv, που ποτέ δεν εκτελείται. Μια λύση στο πρόβλημα είναι η μία διεργασία να εκτελέσει την εντολή recv πρώτα, και στη συνέχεια την εντολή send, ενώ η άλλη διεργασία εκτελεί πρώτα την εντολή send, και μετά την εντολή recv. Για παράδειγμα, σε διάταξη σωλήνωσης, το αδιέξοδο μπορεί να αποφευχθεί,

αν οι «ζυγές» διεργασίες εκτελούν την εντολή send πρωτα και οι «μονές» διεργασίες εκτελούν την εντολή recv πρώτα.

Επειδή η ανταλλαγή δεδομένων μεταξύ δύο διεργασιών είναι συχνή επικοινωνία, το MPI παρέχει την MPI_Sendrecv(), που στέλνει και λαμβάνει δεδομένα και αποκλείει την πιθανότητα αδιεξόδου. Το προηγούμενο παράδειγμα του τοπικού συγχρονισμού μπορεί να υλοποιηθεί με την εντολή MPI_Sendrecv() ως εξής:

```
Process P_{i-1}        Process P_i              Process P_{i+1}

sendrecv(P_i); <-->    sendrecv(P_{i-1});
                       sendrecv(P_{i+1}); <-->  sendrecv(P_i);
```

Σχήμα 6.8:

6.4 Κατηγορίες συγχρονισμένου υπολογισμού

Οι συγχρονισμένοι υπολογισμοί μπορούν να χωριστούν σε δύο κατηγορίες:

- Πλήρως συγχρονισμένοι υπολογισμοί
- Τοπικά συγχρονισμένοι υπολογισμοί

Στους πλήρως συγχρονισμένους υπολογισμούς, όλες οι διεργασίες που εμπλέκονται στον υπολογισμό πρέπει να συγχρονίζονται.

Στους τοπικά συγχρονισμένους υπολογισμούς, οι διεργασίες πρέπει να συγχρονίζονται μόνο με ένα σύνολο λογικά «κοντινών» διεργασιών, και όχι με όλες τις διεργασίες που εμπλέκονται στον υπολογισμό.

6.4.1 Πλήρως συγχρονισμένοι υπολογισμοί

Παράλληλοι Υπολογισμοί Δεδομένων (Data Parallel Computations)

Οι παράλληλοι υπολογισμοί δεδομένων απαιτούν έμμεσο συγχρονισμό μεταξύ των διεργασιών. Σε αυτούς τους υπολογισμούς, η ίδια λειτουργία εκτελείται σε διαφορετικά δεδομένα ταυτόχρονα, δηλαδή παράλληλα. Αυτός ο τύπος παράλληλου υπολογισμού έχει σημαντικά πλεονεκτήματα επειδή:

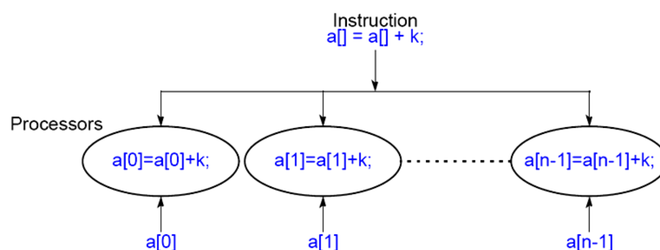
- Είναι εύκολος ο προγραμματισμός σε αυτό το μοντέλο παράλληλης εκτέλεσης. Ουσιαστικά έχουμε ένα μόνο πρόγραμμα.
- Μπορεί εύκολα να κλιμακωθεί σε μεγάλου μεγέθους προβλήματα.
- Πολλοί αριθμητικοί και μη αριθμητικοί υπολογισμοί μπορούν κωδικοποιηθούν ως παράλληλοι υπολογισμοί δεδομένων.

Ακολουθούν παραδείγματα παράλληλων υπολογισμών δεδομένων.

Πρόσθεση σε κάθε στοιχείο ενός πίνακα:

```
for (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

Η πρόταση $a[i] = a[i] + k$ μπορεί να εκτελείται ταυτόχρονα από πολλαπλές διεργασίες, με κάθε μια διεργασία να αναλαμβάνει ένα ξεχωριστό στοιχείο $a[i]$ $(0 < i \leq n)$.



Σχήμα 6.9: Πρόσθεση σταθεράς σε πίνακα με παράλληλο υπολογισμό.

Η δομή forall: Σε πολλές γλώσσες παράλληλου υπολογισμού, ειδικές δομές χρησιμοποιούνται για να δηλώσουν παράλληλους υπολογισμούς δεδομένων. Για παράδειγμα η δομή forall στο κώδικα που ακολουθεί

```
forall (i = 0; i < n; i++) {
    body
}
```

δηλώνει ότι μπορούν να εκτελεστούν ταυτόχρονα ν στιγμιότυπα των προτάσεων που περιέχονται στο body.

Σε κάθε ένα από τα παραπάνω στιγμιότυπα, μία μόνο τιμή της μεταβλητής i είναι έγκυρη. Συγκεκριμένα, στο πρώτο στιγμιότυπο $i = 0$, στο δεύτερο στιγμιότυπο $i = 1$ κοκ. Η διαφορετική τιμή της μεταβλητής i σε κάθε στιγμιότυπο, επιτρέπει διαφοροποίηση της εκτέλεσης των παράλληλων διεργασιών (π.χ. μπορούν να προσπελαύνουν διαφορετικά στοιχεία ενός πίνακα). Για παράδειγμα, για να προσθέσουμε μια σταθερά k σε κάθε στοιχείο ενός πίνακα a, μπορούμε να γράψουμε

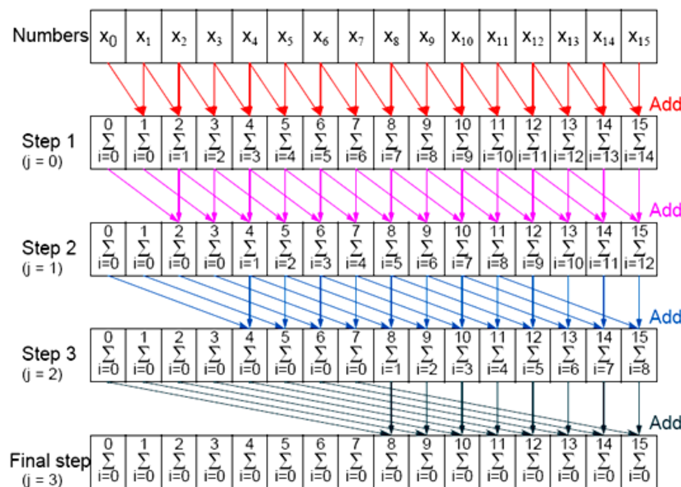
```
forall (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

Όταν η δομή forall υλοποιείται σε συστήματα κατανεμημένης μνήμης, απαιτείται συγχρονισμός μεταξύ των παράλληλων διεργασιών. Για παράδειγμα, στο προηγούμενο παράδειγμα της άθροισης της σταθεράς k στα στοιχεία του πίνακα, η υλοποίηση της forall θα έχει ως εξής:

```
i = myrank;
a[i] = a[i] + k;        /* body */
barrier(mygroup);
```

όπου myrank είναι η ταυτότητα της διεργασίας, ένας αριθμός μεταξύ 0 και $n-1$. Βασική υπόθεση στον παραπάνω κώδικα είναι ότι κάθε διεργασία i έχει πρόσβαση στα αντίστοιχο στοιχείο $a[i]$ του πίνακα.

Υπολογισμός Prefix Sum: Ένα άλλο παράδειγμα παράλληλου υπολογισμού δεδομένων είναι ο υπολογισμός prefix sum. Στο πρόβλημα αυτό, δίνεται μια λίστα από n αριθμούς $x_0, x_1, \ldots, x_{n-1}$ και υπολογίζονται όλα τα μερικά αθροίσματα $s_i = x_0 + x_1 + \cdots + x_i$, $i = 0, 1, \ldots n-1$. Ο ίδιος υπολογισμός μπορεί να ορισθεί και με άλλους αντιμεταθετικούς τελεστές όπως ο πολλαπλασιασμός, το μέγιστο, το ελάχιστο κτλ. Στο Σχήμα 6.9 δίνεται παράδειγμα υπολογισμού των μερικών αθροισμάτων για 16 στοιχεία.



Σχήμα 6.10: Υπολογισμός μερικών αθροισμάτων για 16 στοιχεία.

Γενικά, ο υπολογισμός αυτός απαιτεί $\log n$ βήματα, όπου υπάρχουν n αριθμοί (και n είναι δύναμη του 2). Στο βήμα j $(0 \leq j < \log n)$, εκτελούνται $n - 2j$ προσθέσεις, στις οποίες το στοιχείο $x[i - 2j]$ προστίθεται στο στοιχείο $x[i]$ για $2j \leq i < n$. Ο ακολουθιακός κώδικας για τον υπολογισμό prefix sum είναι ο εξής:

```
for (j=0; j <log n ; j++)
    for (i=2j ; i < n; i++)
        x[i] = x[i] + x[i -2j]
```

Ο παράλληλος κώδικας για τον ίδιο υπολογισμό είναι:

```
for (j=0; j < log n; j++)
    forall (i=0; i < n; i++)
        if (i ? 2j) x[i] = x[i] + x[i ? 2j]
```

Σύγχρονη Επανάληψη (Σύγχρονος Παραλληλισμός): Ο όρος σύγχρονη επανάληψη ή σύγχρονος παραλληλισμός χρησιμοποιείται για να περιγράψει την επίλυση ενός προβλήματος με ένα επαναληπτικό αλγόριθμο, όπου κάθε επανάληψη αποτελείται από ένα πλήθος διεργασιών, οι οποίες ξεκινούν μαζί στην αρχή κάθε επανάληψης. Κάθε επανάληψη δεν μπορεί να αρχίσει μέχρι να έχουν τελειώσει όλες οι διεργασίες την προηγούμενη επανάληψη. Χρησιμοποιώντας τη δομή forall, μια σύγχρονη επανάληψη θα έχει ως εξής:

```
for (j = 0; j < n; j++)           /*for each synch. iteration */
    forall (i = 0; i < N; i++) {  /*N procs each using*/
        body(i);                  /* specific value of i */
    }
```

Η σύγχρονη επανάληψη σε συστήματα κατανεμημένης μνήμης θα έχει ως εξής:

```
for (j = 0; j < n; j++) {       /*for each synchr.iteration */
    i = myrank;                 /*find value of i to be used */
    body(i);
    barrier(mygroup);
}
```

Επίλυση ενός γενικού συστήματος γραμμικών εξισώσεων με επαναληπτική διαδικασία: Υποθέτουμε ότι έχουμε n εξισώσεις γενικής μορφής με n αγνώστους

$$\begin{array}{ccccccccc} a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & \cdots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1} \\ a_{n-2,0}x_0 & + & a_{n-2,1}x_1 & + & \cdots & + & a_{n-2,n-1}x_{n-1} & = & b_{n-2} \\ \vdots & & \vdots & & & & \vdots & \vdots & \vdots \\ a_{0,0}x_0 & + & a_{0,1}x_1 & + & \cdots & + & a_{0,n-1}x_{n-1} & = & b_0 \end{array}$$

όπου οι άγνωστοι είναι οι $x_0, x_1, x_2, \ldots, x_{n-1}$ $(0 \leq i < n)$.

Επιλύοντας την i-οστή εξίσωση ως προς τη μεταβλητή x_i, θα έχουμε:

$$x_i = \frac{1}{a_{i,i}}\left(b_i - \sum_{\substack{i=0 \\ i\neq j}}^{n-1} a_{i,j}x_j\right).$$

Αυτή η εξίσωση, η οποία δίνει τον άγνωστο x_i συναρτήσει των άλλων αγνώστων, μπορεί να χρησιμοποιηθεί επαναληπτικά για κάθε έναν από τους αγνώστους, προκειμένου να πετύχουμε καλύτερη προσέγγιση στις τιμές των αγνώστων. Αυτή η επαναληπτική μέθοδος είναι γνωστή ως **μέθοδος Jacobi**. Σε κάθε επανάληψη της μεθόδου, όλες οι τιμές των x ενημερώνονται μαζί. Μπορεί να αποδειχθεί ότι η μέθοδος Jacobi θα συγκλίνει, αν οι διαγώνιες τιμές των a έχουν απόλυτη τιμή μεγαλύτερη από το άθροισμα των απολύτων τιμών όλων των υπόλοιπων a στη γραμμή (ο πίνακας των a λέγεται σε αυτή την περίπτωση διαγώνια κυρίαρχος (diagonally dominant», δηλαδή αν

$$\sum_{\substack{j=0 \\ j\neq i}}^{n-1} |a_{i,j}| < |a_{i,i}|, \qquad \text{για κάθε } i = 0, 1, \ldots, n-1.$$

Αυτή η συνθήκη είναι ικανή αλλά όχι αναγκαία.

Η επαναληπτική διαδικασία θα πρέπει να τερματίζει όταν έχει επιτευχθεί σύγκλιση στις τιμές των x. Μία απλή προσέγγιση είναι να συγκρίνουμε τις τιμές των x που προέκυψαν στη τρέχουσα επανάληψη με τις τιμές των x από την προηγούμενη επανάληψη. Η επαναληπτική διαδικασία τερματίζεται όταν όλες οι τιμές δεν μεταβάλλονται περισσότερο από ένα προκαθορισμένο όριο $\epsilon > 0$ (error tolerance), δηλαδή όταν

$$|x_i^{(t)} - x_i^{(t-1)}| < \epsilon, \qquad \text{για κάθε } i = 0, 1, \ldots, n-1,$$

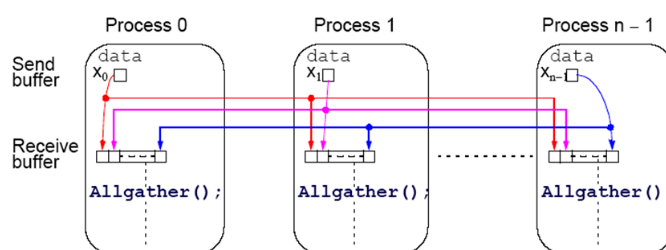
όπου $x_i^{(t)}$ είναι η τιμή του x_i μετά από την t-οστή επανάληψη και $x_i^{(t-1)}$ είναι η τιμή του x_i μετά την $(t-1)$-οστή επανάληψη.

Υποθέτουμε ότι για κάθε άγνωστο υπάρχει μία ξεχωριστή διεργασία που υπολογίζει την τιμή του και ότι κάθε διεργασία θα κάνει το ίδιο πλήθος επαναλήψεων. Μετά το τέλος μίας επανάληψης, οι τρέχουσες τιμές των αγνώστων θα πρέπει να μεταδοθούν σε όλες τις διεργασίες του προγράμματος, αφού κάθε διεργασία πρέπει να γνωρίσει τις τιμές αυτές για να εκτελέσει την επόμενη επανάληψη. Στο τέλος κάθε επανάληψης, πρέπει να υπάρχει ένα φράγμα για να επιτυγχάνεται συγχρονισμός μεταξύ των διεργασιών. Η διεργασία Πι του παράλληλου προγράμματος εκτελεί τον ακόλουθο κώδικα:

```
x[i] = b[i];                                      /*initialize unknown*/
for (iteration = 0; iteration < limit; iteration++) {
    sum = -a[i][i] * x[i];
    for (j = 0; j < n; j++)                  /* compute summation */
```

```
        sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i];      /* compute unknown */
    allgather(&new_x[i]);                   /*bcast/rec values */
    global_barrier();                       /* wait for all procs */
}
```

Με την εντολή allgather(), κάθε διεργασία στέλνει τη νέα τιμή για το $x[i]$ σε κάθε άλλη διεργασία και λαμβάνει τις τιμές όλων των υπολοίπων αγνώστων από τις άλλες διεργασίες. Ανάλογα την υλοποίηση, η εντολή allgather() μπορεί προσφέρει έμμεσο συγχρονισμό στις διεργασίες, αφού μια διεργασία δεν μπορεί να επιστρέψει από αυτή την εντολή, πριν λάβει όλες τις τιμές από τις υπόλοιπες διεργασίες. Σε αυτή την περίπτωση το φράγμα δεν χρειάζεται.



Σχήμα 6.11: Η λειτουργία allgather.

Τα ίδια δεδομένα συλλέγονται από τις διεργασίες. Το δεδομένο από τη διεργασία 0 τοποθετείται πρώτο στους buffers λήψης, το δεδομένο από τη διεργασία 1 τοποθετείται δεύτερο σε αυτούς τους buffers κοκ.

Διαμέριση - Partitioning Στη περιγραφή της παράλληλης υλοποίησης της μεθόδου Jacobi, θεωρήσαμε ότι κάθε διεργασία αναλαμβάνει ένα μόνο στοιχείο εισόδου (άγνωστος x). Συνήθως όμως το πλήθος των επεξεργαστών (διεργασιών) είναι πολύ μικρότερο από το πλήθος των δεδομένων εισόδου του προβλήματος και επομένως οι επεξεργαστές πρέπει να αναλάβουν περισσότερα τους ενός στοιχεία εισόδου. Υπάρχουν δύο τρόποι διαμοιρασμού των αγνώστων στις διαθέσιμες διεργασίες (υποθέτοντας ότι το πλήθος των στοιχείων ν διαιρείται ακριβώς με το πλήθος p των διεργασιών):

- block κατανομή (allocation): Σε αυτή την κατανομή, ομάδες διαδοχικών αγνώστων κατανέμονται στις διεργασίες με αυξανόμενη σειρά. Πιο συγκεκριμένα, η διεργασία P_i αναλαμβάνει τους αγνώστους
$$x_{in/p}, \ldots, x_{(i+1)n/p-1} \qquad i = 0, \ldots, p-1.$$
- κυκλική (cyclic) κατανομή (allocation): Σε αυτή την κατανομή, οι άγνωστοι κατανέμονται κυκλικά στις διεργασίες. Συγκεκριμένα, η διεργασία P_0 αναλαμβάνει τους αγνώστους
$$x_0, x_p, x_{2p}, \ldots, x_{((n/p)-1)p},$$
η διεργασία P_1 αναλαμβάνει τους αγνώστους
$$x_1, x_{p+1}, x_{2p+1}, \ldots, x_{((n/p)-1)p+1}$$
κοκ. Η κυκλική κατανομή δεν έχει κάποιο συγκεκριμένο πλεονέκτημα σε σχέση με τη block κατανομή. Στην πράξη, η κυκλική κατανομή μπορεί να παρουσιάζει δυσκολίες υλοποίησης, αφού απαιτείται πιο πολύπλοκος χειρισμός των δεικτών των αγνώστων.

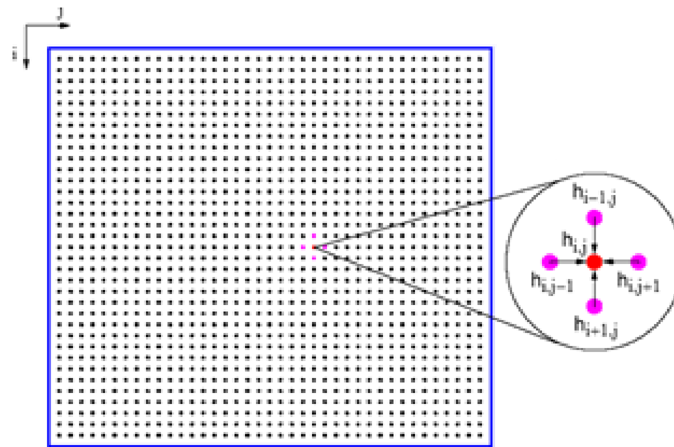
Ανάλυση πολυπλοκότητας της παράλληλης υλοποίησης της μεθόδου Jacobi: Σε κάθε επανάληψη, κάθε μία από τις p συνολικά διεργασίες αναλαμβάνει τον υπολογισμό των τιμών n/p αγνώστων. Για το υπολογισμό της τιμής ενός αγνώστου απαιτούνται $O(n)$ πράξεις. ;ρα συνολικά κάθε διεργασία εκτελεί $O(n2/p)$ πράξεις σε κάθε επανάληψη.

Στο τέλος κάθε επανάληψης υπάρχει η εντολή all-gather και ας υποθέσουμε ότι η υλοποίηση της παρέχει έμμεσο συγχρονισμό στις εμπλεκόμενες διεργασίες και επομένως δεν χρειάζεται η εντολή του φράγματος.

Στη λειτουργία all-gather, κάθε διεργασία στέλνει στις υπόλοιπες διεργασίες τις τιμές των n/p αγνώστων που έχει υπολογίσει στη τρέχουσα επανάληψη. Αν αυτή η επικοινωνία υλοποιηθεί με p διαδοχικές εκπομπές (broadcasts), ο συνολικός χρόνος επικοινωνίας θα είναι $p(t_{startup}+n/pt_{data})$, όπου η παράσταση μέσα στη παρένθεση είναι ο χρόνος για να στείλει μια διεργασία n/p στοιχεία σε όλες τις υπόλοιπες. Γενικά, ο χρόνος εκπομπής εξαρτάται από το διασυνδετικό δίκτυο που συνδέει τους επεξεργαστές. Για t επαναλήψεις της μεθόδου Jacobi, οι παραπάνω χρόνοι πρέπει να πολλαπλασιαστούν επί t.

6.4.2 Τοπικός Σύγχρονος Υπολογισμός

Το πρόβλημα της κατανομής θερμότητας: Στο πρόβλημα αυτό, έχουμε μια περιοχή στην οποία οι τιμές της θερμοκρασίας είναι γνωστές μόνο στις άκρες της. Το ζητούμενο είναι εύρεση των τιμών της θερμοκρασίας και στο εσωτερικό της περιοχής. Για την εύρεση της κατανομής θερμοκρασίας, διαιρούμε την περιοχή, όπως φαίνεται στο Σχήμα 6.12, σε ένα λεπτό πλέγμα σημείων. Έστω $h_{i,j}$ η θερμοκρασία της περιοχής στο σημείο (i, j).

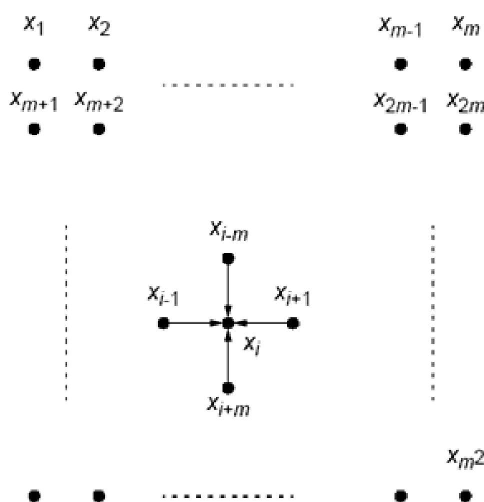


Σχήμα 6.12: Το πρόβλημα της κατανομής θερμότητας.

Γίνεται η υπόθεση ότι η θερμοκρασία σε ένα εσωτερικό σημείο είναι ο μέσος όρος των θερμοκρασιών των τεσσάρων γειτονικών σημείων, δηλαδή

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}.$$

Οι μόνες θερμοκρασίες που γνωρίζουμε είναι οι θερμοκρασίες των εξωτερικών σημείων, δηλαδή τις τιμές των μεταβλητών $h_{i,0}$ και $h_{0,j}$, για $i = 0, 1, \ldots, m$ και για $j = 0, 1, \ldots, m$. Τα εσωτερικά σημεία έχουν συντεταγμένες (i, j), με $0 < i, j < m$. Εφαρμόζοντας επαναληπτικά την παραπάνω σχέση, μπορούμε να υπολογίσουμε τις τιμές $h_{i,j}$, οι οποίες θα συγκλίνουν μετά από μερικές επαναλήψεις.



Σχήμα 6.13:

Αν αριθμήσουμε τα σημεία του πλέγματος από αριστερά προς τα δεξιά και από πάνω προς τα κάτω, τότε η προηγούμενη εξίσωση που δίνει τη θερμοκρασία σε ένα σημείο συναρτήσει των θερμοκρασιών σε γειτονικά σημεία, μπορεί να γραφεί ως εξής:

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-m} + x_{i+m}}{4}.$$

Μπορεί να γραφεί επίσης ως γραμμική εξίσωση των αγνώστων $x_{i-m}, x_{i-1}, x_{i+1}, x_{i+m}$:

$$x_{i-m} + x_{i-1} - 4x_i + x_{i+1} + x_{i+m} = 0.$$

Συνολικά, θα έχουμε ένα αραιό σύστημα με m^2 εξισώσεις και m^2 αγνώστους. Αυτός ο τρόπος επίλυσης είναι γνωστός και ως μέθοδος πεπερασμένων διαφορών (finite difference method).

Χρησιμοποιώντας ένα σταθερό πλήθος επαναλήψεων και χρησιμοποιώντας τη δισδιάστατη αναπαράσταση για τη δεικτοδότηση των αγνώστων, ο ακολουθιακός κώδικας για το πρόβλημα της κατανομής θερμότητας έχει ως εξής:

```
for (iteration = 0; iteration < limit; iteration++) {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
        for (i = 1; i < n; i++)                          /* update points */
            for (j = 1; j < n; j++)
                h[i][j] = g[i][j];
}
```

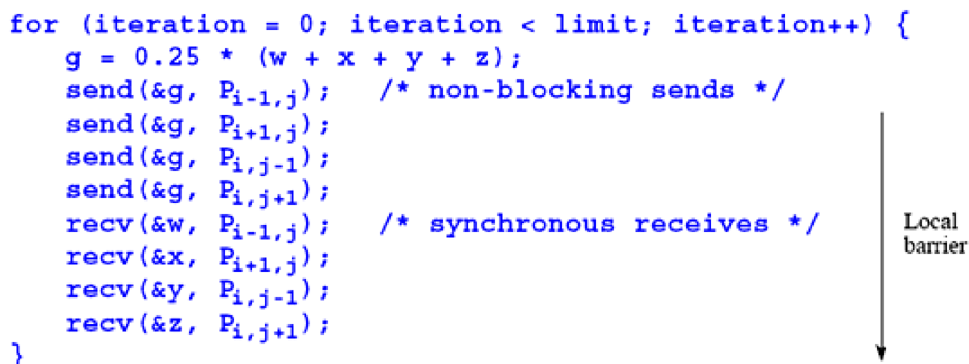
Αν αντίθετα η μέθοδος σταματάει μόλις η επιθυμητή ακρίβεια στις τιμές της θερμοκρασίας επιτευχθεί, ο κώδικας θα έχει ως εξής.

```
do {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
```

```
    for (i = 1; i < n; i++)                          /* update points */
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];

    continue = FALSE; /* indicates whether to continue */
    for (i = 1; i < n; i++)/* check each pt for convergence */
        for (j = 1; j < n; j++)
            if (!converged(i,j) {/* point found not converged */
                continue = TRUE;
                break;
            }
} while (continue == TRUE);
```

Η σύγκλιση ελέγχεται αφότου οι θερμοκρασίες σε όλα τα σημεία έχουν υπολογιστεί. Η ρουτίνα converged(i, j) επιστρέφει TRUE, αν το στοιχείο g[i][j] έχει συγκλίνει στην απαιτούμενη ακρίβεια.

Παράλληλος κώδικας: Θεωρούμε ότι κάθε διεργασία αναλαμβάνει ένα σημείο της επιφάνειας. Έστω $P_{i,j}$ η διεργασία που αναλαμβάνει το σημείο με συντεταγμένες (i, j). Εκτός από τα εξωτερικά σημεία για τα οποία χρειάζεται ειδικός χειρισμός, ο κώδικας που εκτελεί μια διεργασία $P_{i,j}$ που αντιστοιχεί σε ένα εσωτερικό σημείο (i, j) θα έχει ως εξής:

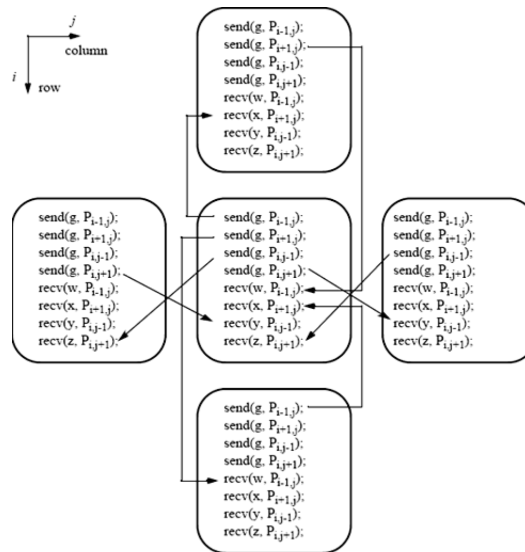


```
for (iteration = 0; iteration < limit; iteration++) {
    g = 0.25 * (w + x + y + z);
    send(&g, P_{i-1,j});   /* non-blocking sends */
    send(&g, P_{i+1,j});
    send(&g, P_{i,j-1});
    send(&g, P_{i,j+1});
    recv(&w, P_{i-1,j});   /* synchronous receives */
    recv(&x, P_{i+1,j});
    recv(&y, P_{i,j-1});
    recv(&z, P_{i,j+1});
}
```


Στον παραπάνω κώδικα, είναι σημαντικό να χρησιμοποιήσουμε send, τα οποία δεν αναστέλλουν την εκτέλεση της διεργασίας περιμένοντας τη λήψη δεδομένων από την αντίστοιχη λειτουργία recv. Σε διαφορετική περίπτωση, θα είχαμε αδιέξοδο, αφού κάθε διεργασία θα περίμενε για ένα recv που ποτέ δεν εκτελείται.

Αντίθετα, για τη λήψη των δεδομένων, πρέπει να χρησιμοποιήσουμε σύγχρονα recv, τα οποία θα περιμένουν για τα αντίστοιχα send. Με τη χρήση σύγχρονων recv επιτυγχάνεται ο συγχρονισμός κάθε διεργασίας με τους τέσσερεις γείτονες.

Για την υλοποίηση της περίπτωσης όπου οι διεργασίες σταματούν όταν φτάνουν στην απαιτούμενη ακρίβεια, πρέπει να υπάρχει μία διεργασία master, η οποία πρέπει να ειδοποιείται όταν οι διεργασίες έχουν σταματήσει. Μια διεργασία μπορεί να στέλνει τα δεδομένα της στη διεργασία master, όταν η επιθυμητή ακρίβεια έχει επιτευχθεί τοπικά. Ο παράλληλος κώδικας θα έχει ως εξής:



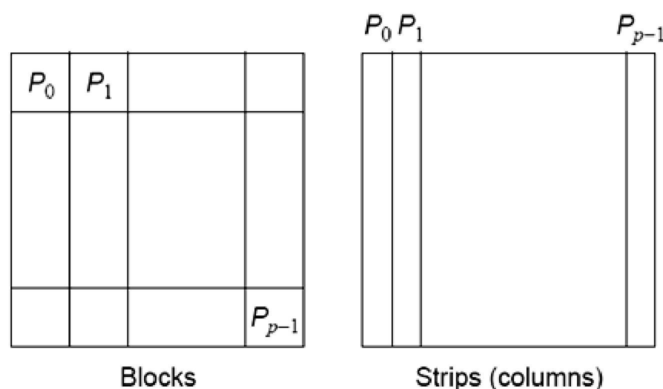
Σχήμα 6.14: Η μεταφορά δεδομένων μεταξύ μίας διεργασίας και των γειτόνων της.

```
iteration = 0;
do {
   iteration++;
   g = 0.25 * (w + x + y + z);
   send(&g, P_{i-1,j});        /* locally blocking sends */
   send(&g, P_{i+1,j});
   send(&g, P_{i,j-1});
   send(&g, P_{i,j+1});
   recv(&w, P_{i-1,j});        /* locally blocking receives */
   recv(&x, P_{i+1,j});
   recv(&y, P_{i,j-1});
   recv(&z, P_{i,j+1});
} while((!converged(i, j)) || (iteration < limit));
send(&g, &i, &j, &iteration, P_{master});
```

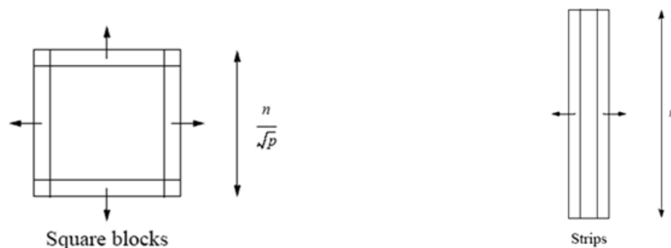
Για να χειριστούμε τις διεργασίες που αναλαμβάνουν εξωτερικά σημεία, μπορούμε να χρησιμοποιήσουμε τις συντεταγμένες της διεργασίας $P_{i,j}$ για να προσδιορίσουμε τη θέση της. Ο κώδικας έχει ως εξής:

```
if (last_row) w = bottom_value;
if (first_row) x = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
iteration = 0;
do {
  iteration++;
  g = 0.25 * (w + x + y + z);
  if !(first_row) send(&g, P_{i-1,j});
  if !(last_row) send(&g, P_{i+1,j});
  if !(first_column) send(&g, P_{i,j-1});
  if !(last_column) send(&g, P_{i,j+1});
  if !(last_row) recv(&w, P_{i-1,j});
  if !(first_row) recv(&x, P_{i+1,j});
  if !(first_column) recv(&y, P_{i,j-1});
  if !(last_column) recv(&z, P_{i,j+1});
} while((!converged) || (iteration < limit));
send(&g, &i, &j, iteration, P_master);
```

Διαμέριση: Στις περισσότερες περιπτώσεις, σε κάθε διεργασία αντιστοιχούν περισσότερα από ένα σημεία, αφού το πλήθος των σημείων της υπό μελέτης περιοχής είναι πολύ μεγαλύτερο από το πλήθος των διαθέσιμων διεργασιών. Τα σημεία της περιοχής μπορούν να χωριστούν σε τετράγωνες περιοχές (blocks) ή σε κατακόρυφες λωρίδες, όπως φαίνεται στο Σχήμα 6.15. Και στις δύο περιπτώσεις, κάθε διεργασία αναλαμβάνει n^2/p στοιχεία.



Σχήμα 6.15:



Σχήμα 6.16:

Στη block διαμέριση, κάθε διεργασία ανταλλάσει δεδομένα από τέσσερεις πλευρές. Αν τα δεδομένα κάθε πλευράς τοποθετούνται σε ένα μήνυμα, κάθε διεργασία στέλνει/λαμβάνει 4 συνο-

λικά μηνύματα από τις τέσσερεις γειτονικές διεργασίες. Το πλήθος των στοιχείων κάθε πλευράς είναι $n/\sqrt{p}$. Συνολικά, ο χρόνος επικοινωνίας σε μία επανάληψη είναι:

$$t_{comm} = 4\left(t_{startup} + \frac{n}{\sqrt{p}} t_{data}\right).$$

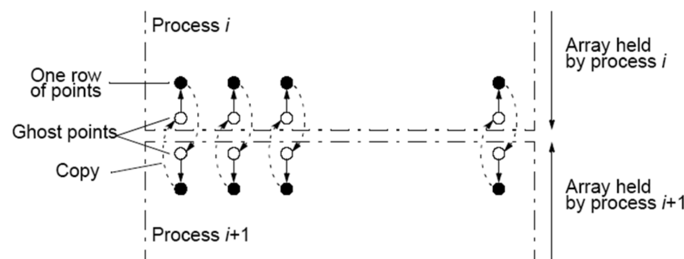
Στη διαμέριση σε λωρίδες, κάθε διεργασία ανταλλάσει δεδομένα από δύο πλευρές. Το πλήθος των στοιχείων κάθε πλευράς είναι n και επομένως ο συνολικός χρόνος επικοινωνίας θα είναι:

$$t_{comm} = 2\left(t_{startup} + n t_{data}\right).$$

Γενικά, η διαμέριση σε λωρίδες προτιμάται, όταν ο χρόνος έναρξης $t_{startup}$ είναι σχετικά μεγάλος, ενώ η block διαμέριση είναι προτιμότερη στην αντίθετη περίπτωση. Με βάση τις προηγούμενες εξισώσεις, η block διαμέριση έχει μεγαλύτερο χρόνο επικοινωνίας από την διαμέριση σε λωρίδες αν

$$t_{startup} > n\left(1 - \frac{2}{\sqrt{p}}\right) t_{data}.$$

Additional row of points at each edge that hold values from adjacent edge. Each array of points increased to accommodate ghost rows.



Σχήμα 6.17: Ghost points.

Ασφάλεια και Αδιέξοδο: Στον κώδικα που έχουμε περιγράψει, όλες οι διεργασίες εκτελούν πρώτα όλες τις εντολές send και στη συνέχεια όλες τις λειτουργίες recv. Αυτός ο τρόπος επικοινωνίας δεν αποκλείει τη πιθανότητα αδιεξόδου. Αυτό οφείλεται στο ότι η ολοκλήρωση της λειτουργίας send εξαρτάται από το διαθέσιμο αποθηκευτικό χώρο που υπάρχει κατά τη στιγμή της εκτέλεσης της send. Αν δεν υπάρχει επαρκής διαθέσιμος χώρος, η λειτουργία send καθυστερεί, μέχρι ο διαθέσιμος χώρος να αυξηθεί ή μέχρι το μήνυμα να μπορεί να σταλεί κατευθείαν, χωρίς να χρειάζεται εκ των προτέρων αποθήκευση. Σε αυτή τη περίπτωση η λειτουργία send λειτουργεί ως συγχρονισμένη λειτουργία και επιστρέφει μόνο όταν το αντίστοιχο recv εκτελείται. Αφού όμως το αντίστοιχο recv δεν εκτελείται ποτέ, συμβαίνει αδιέξοδο.

Ένας τρόπος για να αποφύγουμε το παραπάνω πρόβλημα είναι να εναλλάξουμε τη σειρά των send και recv σε γειτονικές διεργασίες, έτσι ώστε μόνο μία διεργασία εκτελεί πρώτα τις εντολές send. Σε αυτή την περίπτωση, οι λειτουργίες send δεν μπορούν να προκαλέσουν αδιέξοδο.

Εναλλακτικά το MPI προσφέρει παραλλαγές της βασικής send και recv, με τις οποίες αποφεύγουμε την εμφάνιση αδιεξόδου:

- Η εντολή MPI_Sendrecv(), η οποία συνδυάζει την αποστολή και τη λήψη και εγγυάται ότι την απουσία αδιεξόδου.
- Η εντολή MPI_Bsend(), όπου χρήστης παρέχει ο ίδιος αποθηκευτικό χώρο για την προσωρινή αποθήκευση του προς αποστολή μηνύματος.

- Οι μη αναστέλλουσες εντολες MPI_Isend() και MPI_Irecv(), οι οποίες επιστρέφουν κατευθείαν.

  Ξεχωριστές εντολές χρησιμοποιούνται για να διαπιστωθεί αν η λειτουργία που άρχισε με την MPI_Isend() και την MPI_Irecv() έχει ολοκληρωθεί:

  - MPI_Wait(), MPI_Waitall(), MPI_Waitany(): Οι εντολές αυτές περιμένουν μέχρι να ολοκληρωθεί η λειτουργία/ες send ή recv που είναι σε εξέλιξη.
  - MPI_Test(), MPI_Testall(), MPI_Testany(): Οι εντολές αυτές απλά ελέγχουν αν έχει ολοκληρωθεί η λειτουργία send ή recv και επιστρέφουν ανάλογα TRUE ή FALSE.

6.5 Κυψελιδικά Αυτόματα (Cellular Automata)

Σε αυτά τα προβλήματα, ο χώρος του προβλήματος διαιρείται σε κυψέλες. Κάθε κυψέλη μπορεί να είναι σε μία κατάσταση από ένα πεπερασμένο πλήθος καταστάσεων. Οι κυψέλες επηρεάζονται από τους γείτονές τους, σύμφωνα με μερικούς κανόνες, και όλες οι κυψέλες επηρεάζονται ταυτόχρονα ως μία «γενιά». Οι ίδιοι κανόνες επαναλαμβάνονται στις επόμενες γενιές και έτσι με αυτό τον τρόπο οι κυψέλες εξελίσσονται ή αλλάζουν κατάσταση από γενιά σε γενιά.

Τα πιο γνωστό πρόβλημα που μπορεί να περιγραφεί με όρους κυψελιδικού αυτομάτου είναι το «Παιχνίδι της ζωής». Σε αυτό το πρόβλημα υπάρχει μια θεωρητικά άπειρη δισδιάστατη διάταξη από κυψέλες. Κάθε κυψέλη τηρεί έναν «οργανισμό» και έχει οκτώ γειτονικές κυψέλες, συμπεριλαμβανομένων και των διαγώνιων κυψελών. Αρχικά, κάποιες κυψέλες είναι κατειλημμένες. Οι παρακάτω κανόνες εφαρμόζονται:

1. Κάθε οργανισμός με δύο ή τρεις γειτονικούς οργανισμούς επιζεί και στην επόμενη γενιά.
2. Κάθε οργανισμός με τέσσερεις ή περισσότερους γειτονικούς οργανισμούς πεθαίνει λόγω υπερπληθυσμού.
3. Κάθε οργανισμός με ένα ή κανένα γειτονικό οργανισμό πεθαίνει από απομόνωση.
4. Σε κάθε άδεια κυψέλη, γειτονική με ακριβώς τρεις οργανισμούς, δημιουργείται ένας νέος οργανισμός.

Τα κυψελιδικά αυτόματα έχουν εφαρμογές σε πολλά προβλήματα:

- Μηχανική υγρών και αερίων.
- Κίνηση υγρών και αερίων γύρω από αντικείμενα.
- Διάχυση αερίων.
- Προσομοίωση βιολογικών διαδικασιών.
- Προβλήματα αεροδυναμικής, π.χ. η ροή του αέρα σε ένα φτερό αεροπλάνου.
- Διάβρωση/κίνηση του εδάφους σε μία παραλία ή σε μία όχθη ποταμού.

6.6 Μερικώς συγχρονισμένοι υπολογισμοί

Στους υπολογισμούς αυτούς, κάθε διεργασία εκτελείται χωρίς να χρειάζεται να συγχρονίζεται με άλλες διεργασίες μετά από κάθε επανάληψη. Ο μερικός συγχρονισμός επιταχύνει σημαντικά την εκτέλεση του παράλληλου προγράμματος, αφού σε αυτή την περίπτωση, οι διεργασίες συγχρονίζονται σε πιο αραιά διαστήματα και δεδομένου ότι ο συγχρονισμός γενικά είναι μια χρονοβόρα λειτουργία, η οποία επιβραδύνει σημαντικά τον υπολογισμό. Ο καθολικός συγχρονισμός πραγματοποιείται με την εκτέλεση λειτουργιών φράγματος, οι οποίες μερικές φορές έχουν ως αποτέλεσμα την αναίτια καθυστέρηση των διεργασιών.

Το πρόβλημα της κατανομής θερμότητας στο επίπεδο μπορεί επίσης να επιλυθεί με μία τεχνική μερικού συγχρονισμού. Όπως έχουμε αναφέρει, για να λύσουμε το πρόβλημα κατανομής θερμότητας, ο χώρος του προβλήματος διαιρείται σε μια δισδιάστατη διάταξη από σημεία. Η τιμή κάθε σημείου υπολογίζεται από το μέσο όρο των τιμών των τεσσάρων γειτονικών σημείων και η διαδικασία αυτή επαναλαμβάνεται, μέχρι οι τιμές να συγκλίνουν σε μια λύση με ικανοποιητική ακρίβεια. Ο υπολογισμός μπορεί να επιταχυνθεί σημαντικά, αν οι διεργασίες δεν συγχρονίζονται μετά από κάθε επανάληψη.

```
do {
 for (i = 1; i < n; i++)
   for (j = 1; j < n; j++)
     g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);

  for (i = 1; i < n; i++)        /* find max divergence/update pts */
    for (j = 1; j < n; j++) {
     dif = h[i][j] - g[i][j];
     if (dif < 0) dif = -dif;
     if (dif < max_dif) max_dif = dif;
     h[i][j] = g[i][j];
    }
} while (max_dif > tolerance);    /* test convergence */
```

Σχήμα 6.18: Ακολουθιακός κώδικας για την κατανομή θερμότητας.

Αν οι διεργασίες δεν συγχρονίζονται μετά από κάθε επανάληψη, μερικές διεργασίες μπορούν να προχωρήσουν στην επόμενη επανάληψη, πριν όλες οι διεργασίες να έχουν τελειώσει την τρέχουσα επανάληψη. Έτσι, αυτές οι διεργασίες που προχωρούν στην επόμενη επανάληψη χρησιμοποιούν τιμές που έχουν υπολογιστεί, όχι μόνο στη τελευταία επανάληψη, αλλά και σε παλαιές επαναλήψεις. Μετά από αυτές τις αλλαγές, η μέθοδος γίνεται μία ασύγχρονη επαναληπτική μέθοδος.

Στις ασύγχρονες επαναληπτικές μεθόδους, η σύγκλιση σε μια λύση είναι πιο δύσκολη να επιτευχθεί και απαιτούνται αυστηρότερες συνθήκες για να επίτευξή της. Προκειμένου να είναι εφικτή η σύγκλιση, δεν επιτρέπεται γενικά μια διεργασία να χρησιμοποιεί τιμές από αρκετά παλαιές επαναλήψεις. Συγκεκριμένα, έχει αποδειχθεί ότι: Υπάρχει πάντα μια θετική ακέραια σταθερά s τέτοια ώστε, αν κατά τους υπολογισμούς στην i-οστή επανάληψη, κάθε διεργασία δεν χρησιμοποιήσει τιμές που προέκυψαν στην j-οστή επανάληψη, όπου $j < i - s$, η σύγκλιση της μεθόδου είναι εφικτή (Baudet, 1978).

Με βάση την παραπάνω αρχή, επιτρέπεται κάθε διεργασία να συγχρονίζεται με τις γειτονικές διεργασίες, κάθε s επαναλήψεις. Στις ενδιάμεσες επαναλήψεις, κάθε διεργασία χρησιμοποιεί για κάθε γειτονικό σημείο όποια τιμή έχει εκείνη τη στιγμή στη διάθεσή της. Έτσι, οι τιμές των γειτονικών σημείων που χρησιμοποιούνται για την ενημέρωση τιμής ενός σημείου μπορεί να προέρχονται από διαφορετικές επαναλήψεις. Η πιο παλαιά από αυτές μπορεί να απέχει το πολύ s επαναλήψεις από την τρέχουσα. Επίσης, κάθε s επαναλήψεις ελέγχεται αν η λύση έχει

αποκτήσει την επιθυμητή ακρίβεια, δηλαδή έχει συγκλίνει. Αυτή η ασύγχρονη επαναληπτική διαδικασία είναι γνωστή επίσης και με τον όρο χαοτική χαλάρωση (chaotic relaxation).

Κεφάλαιο 7

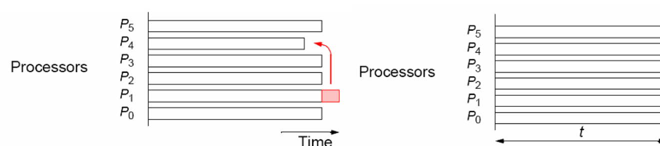
Εξισορρόπηση φόρτου και ανίχνευση τερματισμού

Η εξισορρόπηση φόρτου (load balancing) χρησιμοποιείται για να κατανείμουμε τους υπολογισμούς δίκαια στις διάφορες διεργασίες, προκειμένου να επιτευχθεί ο συντομότερος χρόνος εκτέλεσης.

Το πρόβλημα της ανίχνευσης του τερματισμού (termination detection) είναι να ενημερωθούν όλες οι διεργασίες του παράλληλου προγράμματος ότι ο υπολογισμός έχει ολοκληρωθεί. Το πρόβλημα είναι πιο δύσκολο όταν ο υπολογισμός είναι κατανεμημένος.

Η εξισορρόπηση φόρτου χρησιμοποιείται για να κατανείμουμε τους υπολογισμούς δίκαια στις διάφορες διεργασίες προκειμένου να επιτευχθεί ο συντομότερος χρόνος εκτέλεσης

Το πρόβλημα της ανίχνευσης του τερματισμού είναι να ενημερωθούν όλες οι διεργασίες του παράλληλου προγράμματος ότι ο υπολογισμός έχει ολοκληρωθεί. Το πρόβλημα είναι πιο δύσκολο όταν ο υπολογισμός είναι κατανεμημένος



Σχήμα 7.1: Εξισορρόπηση φόρτου.

Στο αριστερό σχήμα, δεν έχουμε ισοκατανομή φορτίου στις διεργασίες του παράλληλου προγράμματος. Αυτό έχει ως αποτέλεσμα, η διεργασία P_4 να τελειώνει νωρίτερα από όλες τις υπόλοιπες διεργασίες, ενώ η διεργασία P_1 είναι η πιο φορτωμένη από όλες, με αποτέλεσμα να τελειώνει αργότερα από όλες τις άλλες. Παρατηρείστε ότι ο συνολικός χρόνος εκτέλεσης του παράλληλου προγράμματος καθορίζεται από το χρόνο εκτέλεσης της P_1.

Στο δεξί σχήμα, έχουμε ισοκατανομή του φορτίου στις διάφορες διεργασίες και έτσι όλες οι διεργασίες τελειώνουν την ίδια χρονική στιγμή. Αυτό έχει ως αποτέλεσμα να μειώνεται και ο συνολικός χρόνος εκτέλεσης του προγράμματος.

7.1 Στατική εξισορρόπηση φόρτου

Η κατανομή των υπολογισμών στις διάφορες διεργασίες γίνεται πριν την εκτέλεση του παράλληλου προγράμματος. Η στατική εξισορρόπηση φόρτου συνήθως αναφέρεται και ως πρόβλημα απεικόνισης (mapping problem) ή πρόβλημα δρομολόγησης (scheduling problem). Για την

εφαρμογή αυτής της μεθόδου, θα πρέπει να υπάρχει εκτίμηση του χρόνου εκτέλεσης των διαφόρων τμημάτων του προγράμματος και των αλληλεξαρτήσεων μεταξύ των τμημάτων.

Ακολουθούν κάποιες στατικές μέθοδοι εξισορρόπησης φόρτου που έχουν προταθεί:

- Ο αλγόριθμος round robin: Ο αλγόριθμος αυτός μοιράζει τα έργα κυκλικά στις διεργασίες του προγράμματος επιστρέφοντας ξανά στη πρώτη διεργασία όταν δώσει έργα σε όλες τις διεργασίες.

- Τυχαιοκρατικοί (randomized) αλγόριθμοι: μοιράζουν με τυχαίο τρόπο τα έργα στις διαθέσιμες διεργασίες.

- Αναδρομική διχοτόμηση (recursive bisection): Η μέθοδος αυτή διαιρεί αναδρομικά το πρόβλημα σε υποπροβλήματα ίσου υπολογιστικού φόρτου, ελαχιστοποιώντας ταυτόχρονα την επικοινωνία μεταξύ των διεργασιών.

Κατά τη φάση του προσδιορισμού των έργων που θα αναλάβει κάθε διεργασία, είναι βασικό να λαμβάνεται υπόψη το διασυνδετικό δίκτυο που συνδέει τους επεξεργαστές στα συστήματα κατανεμημένης μνήμης. Οι διεργασίες που επικοινωνούν συχνά θα πρέπει να συνδέονται απευθείας προκειμένου να μειωθεί ο χρόνος επικοινωνίας. Το πρόβλημα αυτό στη γενική του μορφή δεν έχει λύση πολυωνυμικού χρόνου. Η στατική μέθοδος εξισορρόπησης φορτίου έχει σημαντικά μειονεκτήματα:

- Είναι πολύ δύσκολο να υπολογιστούν με ακρίβεια οι χρόνοι εκτέλεσης των διάφορων τμημάτων του προγράμματος εκ των προτέρων, χωρίς δηλαδή να εκτελεστούν τα τμήματα αυτά.

- Είναι σχετικά δύσκολο να προσδιορισθεί εκ των προτέρων η καθυστέρηση που προκύπτει από την επικοινωνία μεταξύ των διεργασιών.

- Σε μερικά προβλήματα, ο αριθμός των βημάτων που απαιτούνται για να φτάσουμε στη λύση δεν μπορεί να προσδιορισθεί εκ των προτέρων. Για παράδειγμα, στους αλγόριθμους διάσχισης γραφημάτων δεν είναι γνωστό εκ των προτέρων πόσα πολλά μονοπάτια θα διασχίσει ο αλγόριθμος.

7.2 Δυναμική εξισορρόπηση φόρτου

Στη δυναμική εξισορρόπηση φόρτου, τα διαθέσιμα έργα κατανέμονται στις διεργασίες του παράλληλου προγράμματος κατά την εκτέλεση του προγράμματος. Αυτή η δυναμική κατανομή του φόρτου κατά την ώρα της εκτέλεσης, δημιουργεί πρόσθετη επιβάρυνση αλλά τα οφέλη είναι πιο σημαντικά σε σχέση με τα προβλήματα που μπορεί να φέρει αυτή η τεχνική.

Υπάρχουν δύο βασικές κατηγορίες τεχνικών δυναμικής εξισορρόπησης φόρτου:

- Συγκεντρωτική (centralized).

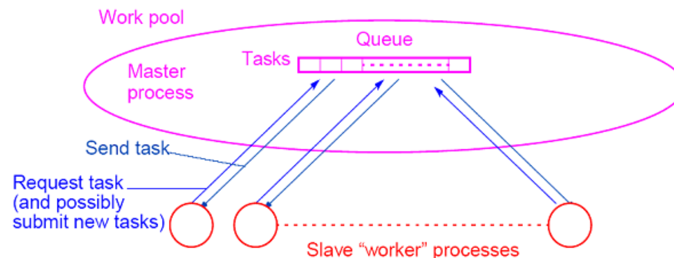
  Στη συγκεντρωτική τεχνική, τα διαθέσιμα προς εκτέλεση έργα δίνονται από μια master διεργασία στις διεργασίες slave του προγράμματος.

- Αποκεντρωμένη (decentralized). Στην αποκεντρωμένη τεχνική, οι διεργασίες δίνουν και παίρνουν διαθέσιμα έργα κατευθείαν η μία από την άλλη χωρίς την επίβλεψη μιας master διεργασίας.

7.2.1 Συγκεντρωτική δυναμική εξισορρόπηση φόρτου

Η master διεργασία διαθέτει τη συλλογή των έργων (work pool) που πρόκειται να εκτελεστούν. Τα έργα στέλνονται στις διεργασίες slave οι οποίες και τα εκτελούν. Όταν μία διεργασία ολοκληρώσει την εκτέλεση ενός έργου, ζητάει από τη master διεργασία ένα νέο προς εκτέλεση έργο. Είναι επίσης πιθανόν οι διεργασίες slave να παράγουν νέα έργα τα οποία υποβάλλονται στη master διεργασία.

Τα διαθέσιμα έργα δεν είναι απαραίτητα να έχουν τον ίδιο υπολογιστικό φόρτο. Σε αυτή την περίπτωση, θα ήταν προτιμότερο να δοθούν πρώτα στις διεργασίες τα «βαρύτερα» έργα και στη συνέχεια τα πιο μικρά γιατί στην αντίθετη περίπτωση, οι διεργασίες που θα έχουν εκτελέσει μικρά έργα θα περιμένουν αδρανείς μέχρι να ολοκληρώσουν την επεξεργασία τους οι διεργασίες που έχουν αναλάβει τα βαρύτερα έργα. Επίσης η ουρά που κρατάει τα προς εκτέλεση έργα δεν χρειάζεται να έχει τη λογική FIFO αλλά μπορεί να είναι μία ουρά προτεραιότητας. Έργα που ενδέχεται να μας οδηγήσουν ταχύτερα στην λύση του υπό μελέτη προβλήματος μπορούν να εκτελούνται νωρίτερα από όλα τα υπόλοιπα.



Σχήμα 7.2: Συγκεντρωτική δυναμική εξισορρόπηση φόρτου.

7.2.2 Τερματισμός

Με τη συγκεντρωτική τεχνική, είναι σχετικά απλό για τη master διεργασία να διαπιστώσει πότε έχει ολοκληρωθεί η εκτέλεση του παράλληλου προγράμματος. Συγκεκριμένα, θα πρέπει να ικανοποιούνται δύο συνθήκες:

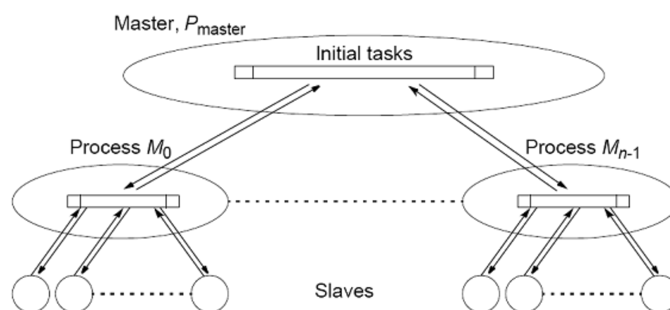
- Η ουρά των έργων είναι άδεια.
- Κάθε slave διεργασία είναι αδρανής και επιπλέον κανένα νέο έργο δεν έχει δημιουργηθεί από κάποια διεργασία.

Δεν αρκεί γενικά να τερματίσουμε τον υπολογισμό όταν η ουρά των έργων είναι άδεια, αφού μία ή περισσότερες διεργασίες μπορεί ακόμα να εκτελούν υπολογισμούς ή ενδέχεται να παράξουν νέα έργα προς εκτέλεση. Υπάρχουν βέβαια προβλήματα στα οποία δεν παράγονται νέα έργα κατά την εκτέλεση του υπολογισμού. Ένα τέτοιο παράδειγμα είναι ο υπολογισμός του fractal Mandelbrot. Σε αυτή την περίπτωση, ο υπολογισμός μπορεί να τερματισθεί όταν η ουρά των έργων μείνει άδεια και όλες οι slave διεργασίες έχουν τερματίσει.

7.2.3 Αποκεντρωμένη Δυναμική Εξισορρόπηση Φόρτου

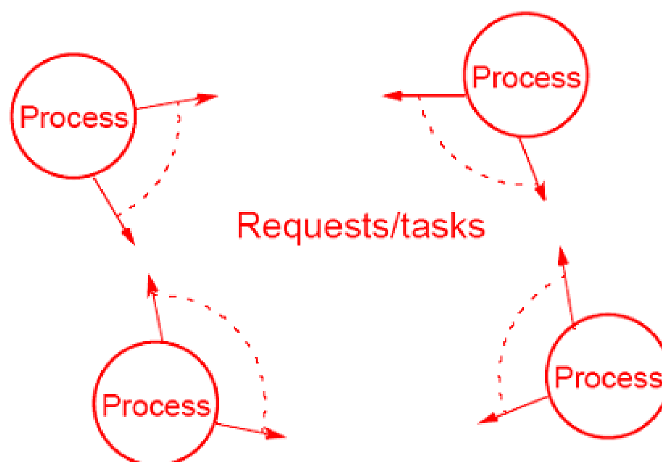
Κατανεμημένη Δεξαμενή Έργων (Distributed Work Pool): Το βασικό πρόβλημα στη συγκεντρωτική τεχνική είναι ότι η master διεργασία μπορεί να δίνει ένα έργο τη φορά στις διεργασίες slave Αυτό μπορεί να δημιουργήσει σοβαρή καθυστέρηση στην εκτέλεση του παράλληλου προγράμματος, όταν πολλές slave διεργασίες απαιτούν έργα ταυτόχρονα από τη master διεργασία.

Για αυτό το λόγο, η συγκεντρωτική τεχνική είναι κατάλληλη όταν υπάρχουν σχετικά λίγες slave διεργασίες και όταν τα έργα είναι σχετικά μεγάλα με επαρκή υπολογιστικό φόρτο. Όταν τα έργα είναι σχετικά μικρά και οι slave διεργασίες είναι πολλές, είναι προτιμότερη η κατανομή της δεξαμενής έργων σε περισσότερες από μία διεργασίες. Μία πιθανή λύση είναι να υπάρχουν ένα σύνολο «μικρότερων» master που αναλαμβάνουν μία υποομάδα των slave διεργασιών. Η αρχική master διεργασία μοιράζει την αρχική δεξαμενή στις «μικρότερες» master. Σε ένα πρόβλημα βελτιστοποίησης, οι μικρότερες masters μπορεί να υπολογίζουν τοπικά βέλτιστα και στη συνέχεια η master να υπολογίζει το συνολικό μέγιστο.



Σχήμα 7.3: Αποκεντρωμένη Δυναμική Εξισορρόπηση Φόρτου.

Πλήρως Κατανεμημένη Δεξαμενή Έργων: Στην τεχνική αυτή, μετά την αρχική κατανομή των έργων στις διεργασίες (κατανεμημένη δεξαμενή), οι διεργασίες ζητούν έργα κατευθείαν από τις άλλες διεργασίες, χωρίς να παρεμβάλλεται η master διεργασία.



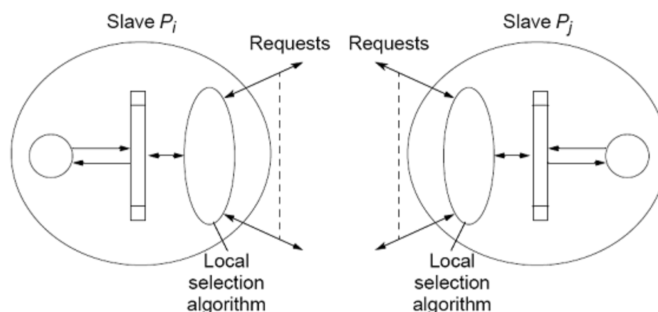
Σχήμα 7.4: Πλήρως Κατανεμημένη Δεξαμενή Έργων.

Τα έργα μεταξύ των διεργασιών μπορούν να μεταφερθούν με δύο μεθόδους:

- Εκκίνηση από τη πλευρά του λήπτη (receiver-initiated).
- Εκκίνηση από τη πλευρά του αποστολέα (sender-initiated).

7.2.4 Μέθοδος εκκίνησης από τη πλευρά του λήπτη

Σε αυτή τη μέθοδο, μια διεργασία απαιτεί έργα από άλλες διεργασίες, όταν η διεργασία έχει λίγα ή δεν έχει κανένα έργο να εκτελέσει. Η μέθοδος αυτή είναι κατάλληλη σε συνθήκες μεγάλου φόρτου στο υπολογιστικό σύστημα.



Σχήμα 7.5: Μέθοδος εκκίνησης από τη πλευρά του λήπτη.

Υπάρχουν πολλές διαφορετικές μέθοδοι για την επιλογή της διεργασίας από την οποία θα ζητηθεί ένα έργο. Δύο απλοί αλγόριθμοι είναι:

- Κυκλική επιλογή (round robin): Η διεργασία P_i ζητά έργα από την διεργασία P_x, όπου ο δείκτης ξ δίνεται από ένα μετρητή, ο οποίος αυξάνεται μετά από κάθε αίτηση, χρησιμοποιώντας modulo n αριθμητική (n διεργασίες), αποκλείοντας την τιμή $x = i$.

- Τυχαία επιλογή: Η διεργασία P_i ζητά έργα από τη διεργασία P_x, όπου ο δείκτης x είναι ένας αριθμός ο οποίος επιλέγεται τυχαία μεταξύ του 0 και $n-1$ (αποκλείοντας την τιμή i).

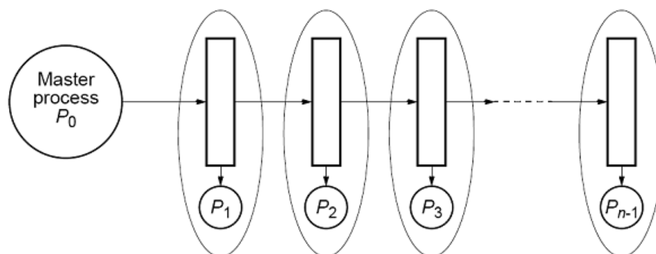
Πάντως, μια επιτυχημένη επιλογή διεργασίας θα πρέπει να λαμβάνει υπόψη και το φόρτο των άλλων διεργασιών και έτσι είναι προτιμότερο να ζητούνται έργα από διεργασίες με υψηλό φόρτο.

7.2.5 Μέθοδος εκκίνησης από τη πλευρά του αποστολέα

Στη μέθοδο αυτή, μια διεργασία με υψηλό φόρτο, στέλνει έργα σε άλλες διεργασίες, οι οποίες είναι πρόθυμες να δεχθούν τα νέα αυτά έργα. Η μέθοδος αυτή αποδίδει κυρίως σε συνθήκες χαμηλού φόρτου στο σύστημα. Για την επιλογή των διεργασιών που θα λάβουν τα νέα έργα, μπορούμε να χρησιμοποιήσουμε παρόμοιες μεθόδους με αυτές της τεχνικής εκκίνησης από την πλευρά του λήπτη. Μπορούμε επίσης να έχουμε ένα συνδυασμό των παραπάνω μεθόδων (sender-initiated και receiver-initiated).

7.2.6 Εξισορρόπηση φορτίου με διεργασίες σε γραμμική διάταξη

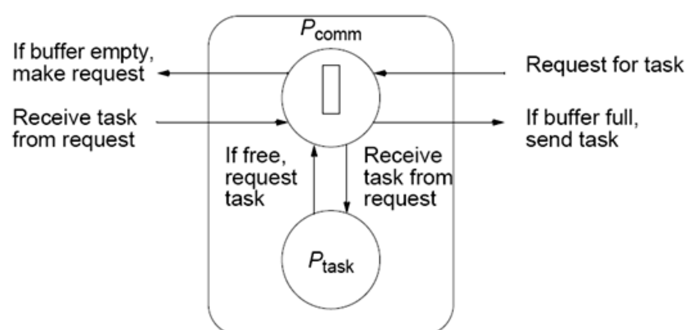
Στο σχήμα 7.6, κάθε διεργασία επικοινωνεί με δύο μόνο διεργασίες (εκτός από την πρώτη και την τελευταία διεργασία που επικοινωνούν με μία μόνο). Η βασική ιδέα είναι να δημιουργήσουμε μια κατανεμημένη ουρά από έργα, με κάθε διεργασία να επιτηρεί ένα διαφορετικό τμήμα της ουράς. Η master διεργασία (P_0) τροφοδοτεί την ουρά με έργα από το ένα άκρο και τα έργα προχωρούν προς τα δεξιά μέσα στην ουρά. Όταν η διεργασία, P_i ($1 \leq i < n$), ανιχνεύσει ένα έργο στο τμήμα της ουράς που ελέγχει και εφόσον η διεργασία είναι αδρανής, παίρνει το έργο από την ουρά και αρχίζει και το εκτελεί. Στη συνέχεια, όλα τα έργα πιο «αριστερά» από αυτό το σημείο, μετακινούνται προς τα δεξιά στην ουρά προκειμένου να καλυφθεί το κενό που δημιουργήθηκε στην ουρά από την εκτέλεση του έργου. Ως τελικό αποτέλεσμα, ένα νέο έργο εισέρχεται στην



Σχήμα 7.6: Εξισορρόπηση φορτίου με διεργασίες σε γραμμική διάταξη.

ουρά από το αριστερό άκρο. Τελικά, όλες οι διεργασίες έχουν ένα έργο να εκτελέσουν και η ουρά συνεχίζει να τροφοδοτείται με νέα έργα. Επίσης, υψηλής προτεραιότητας έργα μπορούν να τοποθετούνται στην ουρά πρώτα, νωρίτερα από όλα τα υπόλοιπα έργα.

Οι λειτουργίες ολίσθησης μπορούν να υλοποιηθούν με την ανταλλαγή μηνυμάτων μεταξύ γειτονικών διεργασιών. Το σχήμα 7.7 δείχνει τις βασικές λειτουργίες που θα πρέπει να εκτελεί κάθε διεργασία.



Σχήμα 7.7:

Ο κώδικας που υλοποιεί την παραπάνω τεχνική εξισορρόπησης φόρτου θα έχει ως εξής:

```
for (i = 0; i < no_tasks; i++) {
    recv(P1, request_tag);       /* request for task */
    send(&task, Pi, task_tag);   /* send tasks into queue */
}
recv(P1, request_tag);           /* request for task */
send(&empty, Pi, task_tag);      /* end of tasks */
```

Σχήμα 7.8: Master process (P_0).

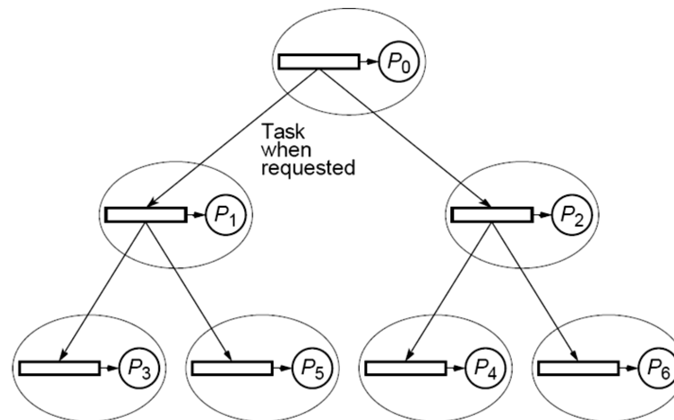
Η μη αναστέλλουσα λειτουργία nrecv() είναι απαραίτητη για να ελέγξουμε αν έχει ληφθεί αίτημα από τα δεξιά. Σε μια υλοποίηση με ΜΠΙ, θα χρησιμοποιούσαμε την συνάρηση MPI_Irecv(), η οποία επιστρέφει αμέσως και «συμπληρώνει» τη δομή request με στοιχεία της λήψης που είναι σε εξέλιξη. Αυτή η δομή χρησιμοποιείται στη συνέχεια, είτε από τη ρουτίνα MPI_Wait(), η οποία περιμένει για την ολοκλήρωση της λήψης, ή από τη ρουτίνα MPI_Test(), η οποία απλά ελέγχει αν έχει ολοκληρωθεί η λήψη.

Η προηγούμενη τεχνική εξισορρόπησης φόρτου μπορεί να επεκταθεί και σε άλλες διατάξεις επικοινωνίας μεταξύ των διεργασιών πέραν της γραμμικής. Στο σχήμα 7.10, οι διεργασίες είναι

```
if (buffer == empty) {
   send(Pi-1, request_tag);     /* request new task */
   recv(&buffer, Pi-1, task_tag)/* task from left proc */
}
if ((buffer == full) && (!busy)) {/* get next task */
   task = buffer;              /* get task*/
   buffer = empty;             /* set buffer empty */
   busy = TRUE;                /* set process busy */
}
nrecv(Pi+1, request_tag, request);/* check msg from right */
if (request && (buffer == full)) {
   send(&buffer, Pi+1);         /* shift task forward */
   buffer = empty;
}
if (busy) {                    /* continue on current task */
   Do some work on task.
   If task finished, set busy to false.
}
```

Σχήμα 7.9: Διεργασία P_i $(1 < i < n)$.

συνδεδεμένες σε ιεραρχική (δενδρική) διάταξη. Το έργο σε κάθε κόμβο περνά σε ένα από τα δύο παιδιά του, συγκεκριμένα αυτό του οποίου ο αποθηκευτικός χώρος δεν περιέχει κάποιο έργο.



Σχήμα 7.10:

7.3 Κατανεμημένη Ανίχνευση Τερματισμού

Όταν ο υπολογισμός είναι κατανεμημένος, η διαπίστωση ότι ο υπολογισμός έχει συνολικά ολοκληρωθεί μπορεί να είναι δύσκολη υπόθεση, εκτός αν το πρόβλημα είναι τέτοιο ώστε μόνο μία από τις διεργασίες φθάνει στη λύση. Γενικά, ο κατανεμημένος τερματισμός τη χρονική στιγμή t θα πρέπει να ικανοποιεί τις ακόλουθες συνθήκες:

- Θα πρέπει να ικανοποιούνται τοπικές συνθήκες τερματισμού σε κάθε διεργασία τη χρονική στιγμή t. Οι συνθήκες τερματισμού σε κάθε διεργασία εξαρτώνται από τη συγκεκριμένη εφαρμογή. Για παράδειγμα, στην επαναληπτική μέθοδο Jacobi, η τοπική συνθήκη τερματισμού σε μία διεργασία συμβαίνει όταν η τιμή του τοπικού αγνώστου έχει συγκλίνει στην επιθυμητή ακρίβεια.

- Δεν υπάρχουν μηνύματα μεταξύ των διεργασιών τη χρονική στιγμή t, τα οποία να είναι υπό μεταφορά.

Η λεπτή διαφορά μεταξύ των παραπάνω συνθηκών τερματισμού και αυτών που δόθηκαν για τον τερματισμό σε ένα σύστημα συγκεντρωτικής εξισορρόπησης φόρτου είναι ότι τώρα λαμβάνονται υπόψη τα μηνύματα τα οποία μπορεί να είναι υπό μεταφορά μεταξύ των διεργασιών. Η δεύτερη συνθήκη είναι απαραίτητη, επειδή ένα μήνυμα υπό μεταφορά μπορεί να επανεκκινήσει μία διεργασία που έχει ήδη τερματίσει. Η πρώτη συνθήκη είναι εύκολο να ανιχνευθεί, αφού κάθε διεργασία μπορεί να στείλει ένα μήνυμα στη master διεργασία αφότου οι τοπικές συνθήκες τερματισμού ικανοποιούνται. Η δεύτερη συνθήκη είναι πιο δύσκολη να ανιχνευθεί, αφού ο χρόνος που απαιτείται για να σταλούν τα μηνύματα μεταξύ των διεργασιών δεν είναι γνωστός εκ των προτέρων και εξαρτάται από τη συγκεκριμένη αρχιτεκτονική στην οποία υλοποιείται ο υπολογισμός. Η πιο απλή λύση είναι να αφήσουμε ένα επαρκές χρονικό διάστημα, προκειμένου τα μηνύματα να παραληφθούν. Όμως αυτό δεν αποτελεί γενική λύση.

Οι Bertsekas και Tsitsiklis πρότειναν μία γενική μέθοδο κατανεμημένου τερματισμού. Κάθε διεργασία μπορεί να είναι σε μία από τις δύο καταστάσεις:

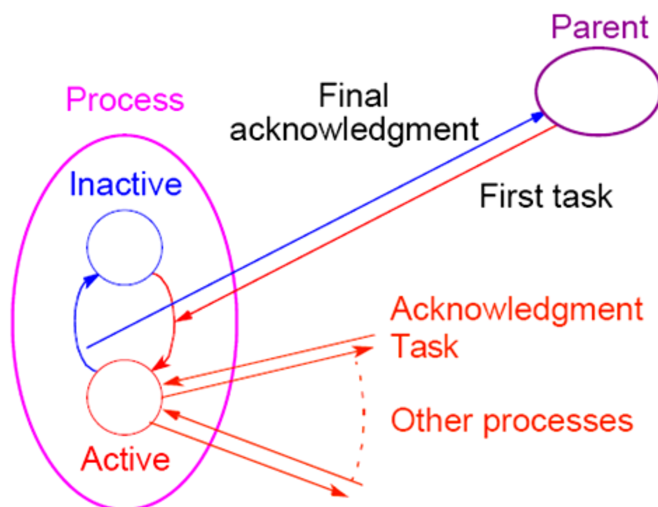
1. Αδρανής: Η διεργασία δεν έχει κάποιο έργο να εκτελέσει.
2. Ενεργή.

Η διεργασία η οποία έστειλε το αρχικό έργο σε μία διεργασία και την έκανε να αλλάξει κατάσταση και να γίνει ενεργή, γίνεται πατέρας αυτής της διεργασίας.

Όταν η διεργασία λαμβάνει ένα έργο, στέλνει αμέσως ένα μήνυμα επιβεβαίωσης λήψης (acknowledgment message), εκτός αν το έργο που λαμβάνει η διεργασία προέρχεται από τη διαδικασία πατέρα. Μια διεργασία στέλνει μήνυμα ack στον πατέρα της, μόνο όταν η διεργασία είναι έτοιμη να γίνει αδρανής, δηλαδή όταν:

- Η τοπική συνθήκη τερματισμού ικανοποιείται (όλα τα έργα έχουν ολοκληρωθεί) και
- έχει στείλει όλα τα μηνύματα ack για τα έργα που έχει λάβει, και
- έχει λάβει όλα τα μηνύματα ack για τα έργα που έχει στείλει σε άλλες διεργασίες.

Η τελευταία συνθήκη έχει ως πρακτικό αποτέλεσμα μια διεργασία να γίνεται αδρανής πάντα πριν τη διαδικασία πατέρα της. Όταν η πρώτη διεργασία που ενεργοποιήθηκε στο πρόγραμμα γίνει αδρανής, ο υπολογισμός μπορεί να τερματίσει.



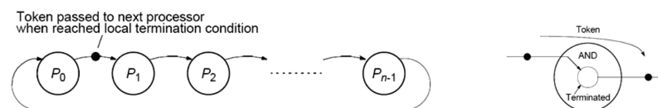
Σχήμα 7.11: Τερματισμός χρησιμοποιώντας μηνύματα επιβεβαίωσης λήψης μηνύματος.

7.3.1 Αλγόριθμος Ανίχνευσης Τερματισμού με διεργασίες σε διάταξη δακτυλίου ενός περάσματος

Όταν οι διεργασίες επικοινωνούν σε διάταξη δακτυλίου, μπορεί να εφαρμοσθεί ο ακόλουθος αλγόριθμος ανίχνευσης τερματισμού:

- Όταν η διεργασία P_0 έχει τερματίσει, παράγει ένα τεκμήριο (token) και το περνάει στη διεργασία P_1.
- Όταν η διεργασία P_i ($1 \leq i < n$) λάβει το τεκμήριο και έχει ήδη τερματίσει, περνάει το τεκμήριο στην διεργασία P_{i+1}. Διαφορετικά, περιμένει μέχρι την ικανοποίηση της τοπικής συνθήκης τερματισμού. Από τη στιγμή που η συνθήκη επαληθευθεί, περνάει το τεκμήριο στην επόμενη διεργασία P_{i+1}. Η διεργασία P_{n-1} περνάει το τεκμήριο στη διεργασία P_0.
- Όταν η P_0 λάβει το τεκμήριο, γνωρίζει ότι όλες οι διεργασίες στο δακτύλιο έχουν τερματίσει. Αν χρειάζεται, η διεργασία P_0 μπορεί να στέλνει ένα μήνυμα σε όλες τις διεργασίες, πληροφορώντας τες για τον συνολικό τερματισμό.

Ο συγκεκριμένος αλγόριθμος υποθέτει ότι μία διεργασία δεν πρόκειται να ενεργοποιηθεί ξανά μετά το τοπικό τερματισμό της. Έτσι, η τεχνική αυτή δεν μπορεί να εφαρμοσθεί σε περιπτώσεις που μία διεργασία στέλνει ένα νέο έργο σε μία ήδη αδρανή διεργασία.



Σχήμα 7.12:

7.3.2 Αλγόριθμος Τερματισμού δύο περασμάτων με τις διεργασίες σε διάταξη δακτυλίου

Μπορούμε να τροποποιήσουμε την προηγούμενη μέθοδο, έτσι ώστε να προβλέπουμε και την περίπτωση της επανεργοποίησης μιας διεργασίας. Η νέα μέθοδος απαιτεί δύο περάσματα του τεκμηρίου γύρω από το δακτύλιο. Ο δεύτερος γύρος μπορεί να χρειάζεται γιατί η διεργασία P_i ενδέχεται να περάσει ένα έργο στη διεργασία P_j, όπου $j < i$, αφότου το τεκμήριο έχει περάσει από την P_j. Αν αυτό λοιπόν συμβεί, το τεκμήριο πρέπει να περάσει από όλες τις διεργασίες δεύτερη φορά.

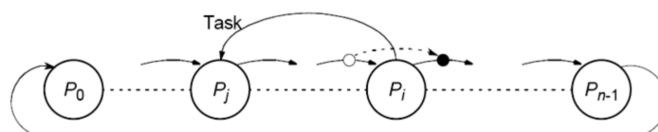
Ο τρόπος με τον οποίο οι διεργασίες μπορούν να καταλάβουν ότι ένας δεύτερος γύρος είναι απαραίτητος είναι ο «χρωματισμός» του τεκμηρίου με δύο χρώματα, μαύρο ή άσπρο, ανάλογα με το αν χρειάζεται δεύτερος γύρος ή όχι. Επίσης και οι διεργασίες χρωματίζονται ως μαύρες ή άσπρες. Η λήψη ενός μαύρου τεκμηρίου από μια διεργασία σημαίνει ότι συνολικός τερματισμός δεν έχει επιτευχθεί και ότι το τεκμήριο θα πρέπει να κυκλοφορήσει πάλι στο δακτύλιο.

Αναλυτικά, ο αλγόριθμος έχει ως εξής:

- Η P_0 γίνεται λευκή, όταν τερματίζει, και παράγει ένα λευκό τεκμήριο που το στέλνει στη διεργασία P_1.
- Κάθε διεργασία, αφότου τερματίσει, περνάει το τεκμήριο που έχει λάβει στην επόμενη διεργασία. Όμως το χρώμα του τεκμηρίου μπορεί να αλλάξει. Συγκεκριμένα, αν η διεργασία P_i περάσει ένα έργο στη διεργασία P_j, όπου $j < i$ (δηλαδή σε μία διεργασία πιο αριστερά από αυτή στο δακτύλιο), γίνεται μαύρη διεργασία, αλλιώς παραμένει λευκή.

Μια μαύρη διεργασία θα χρωματίσει το τεκμήριο μαύρο και θα το περάσει στην επόμενη διεργασία. Μία άσπρη διεργασία θα διατηρήσει το χρώμα του τεκμηρίου που έλαβε, περνώντας το στην επόμενη χωρίς αλλαγές. Αφότου η διεργασία P_i περάσει το τεκμήριο, η P_i ξαναγίνεται άσπρη.

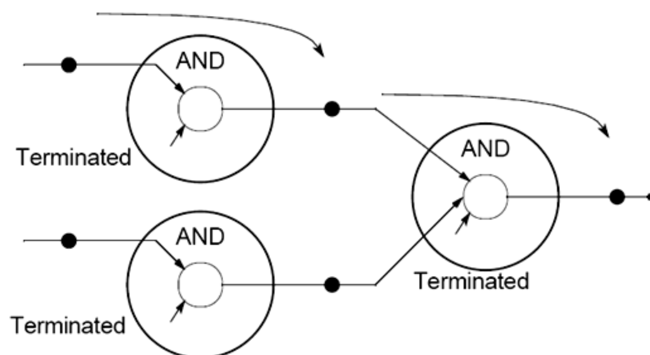
- Όταν η διεργασία P_0 λάβει ένα μαύρο τεκμήριο, το περνάει στην επόμενη διεργασία, αφού πρώτα το χρωματίσει άσπρο. Αν η διεργασία P_0 λάβει ένα άσπρο τεκμήριο, σημαίνει ότι όλες οι διεργασίες έχουν τερματίσει.



Σχήμα 7.13:

7.3.3 Αλγόριθμος τερματισμού με διεργασίες σε διάταξη δένδρου

Η παραπάνω τεχνική μπορεί να γενικευθεί με τις διεργασίες σε διάταξη δένδρου. Σε αυτή την περίπτωση, μία διεργασία περνάει το τεκμήριο στον πατέρα της, αν έχουν ληφθεί τεκμήρια και από τα δύο παιδιά της. Αυτό πρακτικά σημαίνει ότι ο υπολογισμός έχει ολοκληρωθεί σε όλο το υποδέντρο με ρίζα τη συγκεκριμένη διεργασία.



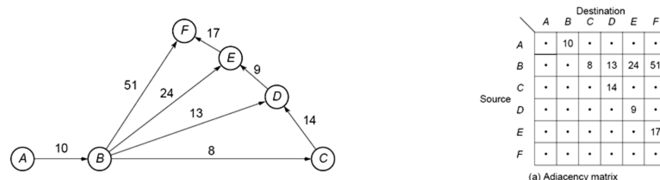
Σχήμα 7.14:

7.3.4 Αλγόριθμος κατανεμημένου τερματισμού με σταθερή ενέργεια

Στον αλγόριθμο αυτό, χρησιμοποιείται μία σταθερή ποσότητα, η οποία θεωρείται ως «ενέργεια» του συστήματος. Αυτή η ενέργεια είναι γενίκευση της έννοιας του τεκμηρίου. Το σύστημα ξεκινά με όλη αυτή την ενέργεια συγκεντρωμένη στη διαδικασία ρίζα. Σε κάθε διεργασία που ζητάει έργα να εκτελέσει, η διαδικασία ρίζα δίνει μαζί με τα έργα και μέρος της ενέργειας που διαθέτει στις διεργασίες αυτές. Με παρόμοιο τρόπο, αν μια διεργασία λάβει αιτήσεις για έργα, η ενέργεια μοιράζεται περαιτέρω. Όταν μια διεργασία γίνει αδρανής, δίνει πίσω την ενέργεια που διαθέτει πριν ζητήσει νέο έργο. Αυτή η ενέργεια μπορεί να επιστραφεί κατευθείαν πίσω στη διεργασία-ρίζα ή στη διεργασία που έδωσε το αρχικό έργο στη συγκεκριμένη διεργασία. Στη

δεύτερη περίπτωση, δημιουργείται μία δενδρική ιεραρχία μεταξύ των διεργασιών. Μία διεργασία δεν επιστρέφει την ενέργειά της, μέχρι να συλλέξει πάλι όλη την ενέργεια που έχει μοιράσει. Όταν όλη η ενέργεια έχει επιστραφεί πίσω στη διεργασία ρίζα, όλες οι διεργασίες είναι αδρανείς και ο υπολογισμός μπορεί να τερματισθεί συνολικά.

7.3.5 Παράδειγμα εξισορρόπησης φόρτου και ανίχνευσης τερματισμού



	A	B	C	D	E	F
A	•	10	•	•	•	•
B	•	•	8	13	24	51
C	•	•	•	14	•	•
D	•	•	•	•	9	•
E	•	•	•	•	•	17
F	•	•	•	•	•	•

Σχήμα 7.15:

Το πρόβλημα που θα μελετηθεί είναι η παράλληλη υλοποίηση της εύρεσης των συντομότερων διαδρομών από μία κορυφή (στο συγκεκριμένο παράδειγμα από την κορυφή A) προς όλες τις υπόλοιπες. Υπάρχουν δύο γνωστοί αλγόριθμοι για την εύρεση συντομότερων διαδρομών με κοινή αφετηρία:

- Ο αλγόριθμος Moore (1957).
- Ο αλγόριθμος Dijkstra (1959).

Οι δύο αυτοί αλγόριθμοι είναι παρόμοιοι. Στο συγκεκριμένο παράδειγμα, θα ακολουθήσουμε τον αλγόριθμο του Moore, επειδή είναι πιο εύκολος να παραλληλοποιηθεί αν και κάνει περισσότερους υπολογισμούς από ότι ο αλγόριθμος του Dijkstra. Τα βάρη στο γράφημα θα πρέπει να είναι όλα θετικά προκειμένου η μέθοδος να εφαρμοσθεί.

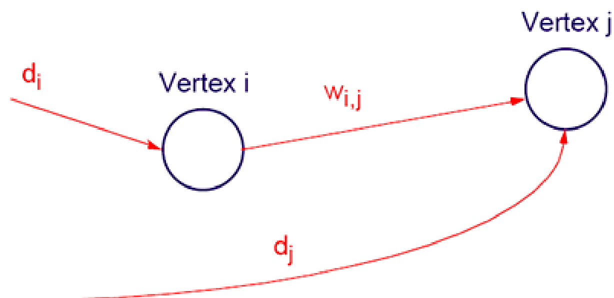
Ο αλγόριθμος του Μοορε:

- Με αρχή τον κόμβο αφετηρία, ο βασικός αλγόριθμος επισκέπτεται διαδοχικά κάθε κορυφή i του γραφήματος και εκτελεί τα εξής: Για κάθε άλλη κορυφή j, ελέγχεται αν η διαδρομή από την αφετηρία μέχρι την κορυφή i, που διέρχεται μέσω της κορυφής i, είναι συντομότερη από αυτή που έχει ευρεθεί μέχρι εκείνη τη στιγμή. Σε αυτή την περίπτωση, κρατείται αυτή ως η ελάχιστη τρέχουσα απόσταση από την αφετηρία προς τη συγκεκριμένη κορυφή j. Πιο αναλυτικά, αν d_i είναι η τρέχουσα ελάχιστη απόσταση από την αφετηρία προς την κορυφή i και $w_{i,j}$ είναι το βάρος της ακμής από την κορυφή i στην κορυφή j, η ενημέρωση που γίνεται σε κάθε βήμα περιγράφεται από την ακόλουθη σχέση

$$d_j = \min\{d_j, d_i + w_{i,j}\}.$$

- Ο παραπάνω αλγόριθμος επαναλαμβάνεται, μέχρι να μην προκύψει καμία αλλαγή στις αποστάσεις d_j, για όλες τις διεργασίες j μετά από μία επανάληψη.

Για την υλοποίηση του αλγορίθμου, χρησιμοποιείται μία ουρά ΦΙΦΟ, η οποία αποθηκεύει τη λίστα των κορυφών που θα εξεταστούν. Αρχικά, μόνο η κορυφή-αφετηρία είναι στην ουρά. Επίσης, οι τρέχουσες μικρότερες αποστάσεις από την αφετηρία στις υπόλοιπες κορυφές αποθηκεύονται στον πίνακα d. Συγκεκριμένα, η τρέχουσα μικρότερη απόσταση προς την κορυφή i είναι αποθηκευμένη στο στοιχείο $d[i]$. Αρχικά, καμία από αυτές τις αποστάσεις δεν είναι



Σχήμα 7.16:

γνωστές και έτσι αρχικοποιούνται στη τιμή άπειρο. Τα βάρη των ακμών του γραφήματος είναι αποθηκευμένα στον πίνακα w και $w[i][j]$ είναι το βάρος της ακμής από την κορυφή i στην κορυφή j. Το βάρος αυτό είναι άπειρο αν δεν υπάρχει ακμή μεταξύ δύο κόμβων. Ο κώδικας που ενημερώνει τις ελάχιστες αποστάσεις σε κάθε βήμα έχει ως εξής:

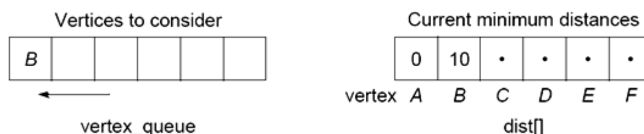
```
newdist_j = dist[i] + w[i][j];
if (newdist_j < dist[j]) dist[j] = newdist_j;
```

Όταν η συντομότερη απόσταση προς την κορυφή j μειώνεται, η κορυφή j προστίθεται στην ουρά (αν δεν είναι στην ουρά). Αυτό έχει ως αποτέλεσμα, η κορυφή j να εξετάζεται ξανά από τον αλγόριθμο.

Στη συνέχεια, ακολουθεί η εκτέλεση του αλγορίθμου για το γράφημα του παραδείγματος

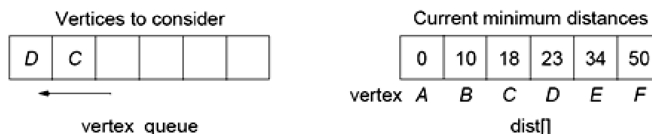
- Η αρχική κατάσταση των δύο δομών δεδομένων έχει ως εξής:

- Μετά την εξέταση της κορυφής A, θα έχουμε:

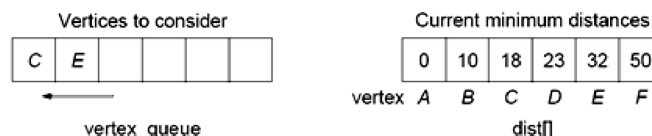


- Μετά την εξέταση των ακμών από την B στην F, E, D, και C έχουμε:

- Μετά την εξέταση της ακμής από την κορυφή E στην F θα έχουμε:



- Μετά την εξέταση της ακμής από την κορυφή D στην E θα έχουμε:



- Μετά την εξέταση της ακμής από την κορυφή C στη D, δεν προκύπτουν αλλαγές. Μετά την εξέταση της ακμής E προς την F θα έχουμε:

- Η ουρά έχει αδειάσει και επομένως δεν υπάρχουν άλλες κορυφές να εξετάσουμε. Ο πίνακας d περιέχει πλέον τις ελάχιστες αποστάσεις από την κορυφή A προς τις υπόλοιπες κορυφές του γραφήματος.

Ακολουθιακός κώδικας για τον αλγόριθμο του Moore: Αν η συνάρτηση next_vertex() επιστρέφει την επόμενη κορυφή από την ουρά των κορυφών ή την τιμή no_vertex, εάν η ουρά είναι κενή, ο ακολουθιακός κώδικας για τον αλγόριθμο του Moore θα έχει ως εξής:

```
while ((i = next_vertex()) != no_vertex)  /* while a vertex */
  for (j = 1; j < n; j++)                /* get next edge */
    if (w[i][j] != infinity) {           /* if an edge */
      newdist_j = dist[i] + w[i][j];
      if (newdist_j < dist[j]) {
        dist[j] = newdist_j;
        append_queue(j);          /* add to queue if not there */
      }
    }                             /*no more to consider*/
```

Σχήμα 7.17:

Παράλληλη υλοποίηση του αλγόριθμου του Moore: Θα παρουσιάσουμε πρώτα μία υλοποίηση με συγκεντρωτική δεξαμενή έργων. Η συγκεντρωτική δεξαμενή έργων είναι ουσιαστικά η ουρά των κορυφών, vertex_queue.

Κάθε slave διεργασία παίρνει κορυφές από την ουρά και επιστρέφει νέες κορυφές, των οποίων οι συντομότερες αποστάσεις έχουν αλλάξει. Αφού, ο πίνακας των βαρών των ακμών του γραφήματος δεν αλλάζει κατά την εκτέλεση του παράλληλου προγράμματος, αυτή η δομή αντιγράφεται από την αρχή της εκτέλεσης σε κάθε slave διεργασία.

Ακολουθεί ο κώδικας για τη master διεργασία.

Ο κώδικας για τη slave διεργασία i είναι ο εξής:

Στον παραπάνω κώδικα, η διεργασία master στέλνει στη slave διεργασία μαζί με την κορυφή που θα επεξεργαστεί και τον πίνακα d με τις τρέχουσες ελάχιστες αποστάσεις.

```
while (vertex_queue() != empty) {
  recv(PANY, source = Pi);   /* request task from slave */
  v = get_vertex_queue();
  send(&v, Pi);              /* send next vertex and */
  send(&dist, &n, Pi);       /* current dist array */
    .
  recv(&j, &dist[j], PANY, source = Pi);/* new distance */
  append_queue(j, dist[j]);  /* append vertex to queue */
                             /* and update distance array */
};
recv(PANY, source = Pi);     /* request task from slave */
send(Pi, termination_tag);   /* termination message*/
```

Σχήμα 7.18:

```
send(Pmaster);               /* send request for task */
recv(&v, Pmaster, tag);      /* get vertex number */
if (tag != termination_tag) {
  recv(&dist, &n, Pmaster);  /* and dist array */
  for (j = 1; j < n; j++)    /* get next edge */
    if (w[v][j] != infinity) {/* if an edge */
      newdist_j = dist[v] + w[v][j];
      if (newdist_j < dist[j]) {
        dist[j] = newdist_j;
        send(&j, &dist[j], Pmaster);/* add vertex to queue */
      }                         /* send updated distance */
    }
}
```

Σχήμα 7.19:

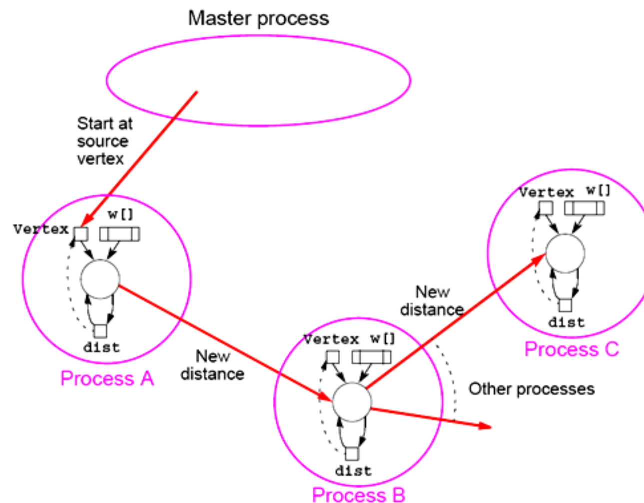
Αποκεντρωμένη Δεξαμενή Έργων: Σε αυτή την τεχνική, η ουρά vertex_queue κατανέμεται στις διεργασίες. Συγκεκριμένα, η slave διεργασία i αναλαμβάνει την επεξεργασία της κορυφής i και επομένως κατέχει την εγγραφή της ουράς για την κορυφή i. Με άλλα λόγια, για κάθε κορυφή i δεσμεύεται μία θέση στην ουρά και αυτή η θέση είναι υπό τον έλεγχο της διεργασίας i.

Ο πίνακας dist[ ] κατανέμεται μεταξύ των διεργασιών κατά τέτοιο τρόπο ώστε η διεργασία i να διατηρεί πάντα την τρέχουσα ελάχιστη απόσταση προς την κορυφή i. Επίσης, κάθε διεργασία i αποθηκεύει την i-οστή γραμμή από το πίνακα γειτνίασης του γραφήματος. Η γραμμή αυτή περιέχει τα βάρη των ακμών της κορυφής i προς όλες τις υπόλοιπες κορυφές του γραφήματος.

Ο αλγόριθμος αναζήτησης έχει ως εξής:

- Η επεξεργασία αρχίζει από την κορυφή A. Η διαδικασία που έχει αναλάβει στην κορυφή A ενεργοποιείται.
- Αυτή η διεργασία θα βρει τις αποστάσεις προς τους γείτονες της κορυφής A.
- Η απόσταση προς την κορυφή j θα σταλεί στη διεργασία j, η οποία θα συγκρίνει τη νέα τιμή με αυτή που έχει ήδη αποθηκευμένη και θα την ενημερώσει, αν η νέα απόσταση είναι μικρότερη από την αποθηκευμένη.
- Έτσι, με αυτό τον τρόπο, όλες οι ελάχιστες αποστάσεις θα ενημερωθούν κατά τη διάρκεια της αναζήτησης.
- Αν το περιεχόμενο του $d[i]$ αλλάξει, η διεργασία i θα ενεργοποιηθεί ξανά και αρχίζει να υπολογίζει τις αποστάσεις προς τους γείτονες της κορυφής i σε ένα νέο γύρο αναζήτησης.

Ο κώδικας για τη slave διεργασία έχει ως εξής:



Σχήμα 7.20:

```
recv(newdist, PANY);
if (newdist < dist) {
    dist = newdist;
    vertex_queue = TRUE;
}
else vertex_queue =  FALSE;
if (vertex_queue == TRUE)
    for (j=0; j <n ; j++)
        if (w[j] != infinity) {
            d = dist + w[j];
            send(&d, Pj);
        }
```

Ο κώδικας για την σλαε διεργασία i μπορεί να απλοποιηθεί ως εξής:

```
recv(newdist, PANY);
if (newdist < dist)
  dist = newdist;          /* start searching around vertex */
  for (j = 1; j < n; j++) /* get next edge */
     if (w[j] != infinity) {
        d = dist + w[j];
        send(&d, Pj);       /* send distance to proc j */
     }
```

Σχήμα 7.21:

Ένας μηχανισμός χρειάζεται για να διαπιστωθεί ο τερματισμός ολόκληρου του υπολογισμού. Συγκεκριμένα, θα πρέπει ο υπολογισμός να τερματίζεται, όταν όλες οι διεργασίες είναι αδρανείς και όταν δεν υπάρχουν μηνύματα υπό μεταφορά. Η απλούστερη λύση είναι να χρησιμοποιηθούν σύγχρονες ρουτίνες επικοινωνίας, όπου μία διεργασία-αποστολέας δεν μπορεί να προχωρήσει, μέχρι ο παραλήπτης να λάβει το μήνυμα που εστάλη. Σημειώνεται επίσης ότι μία διεργασία είναι ενεργή, μόνο αφότου η κορυφή της έχει τοποθετηθεί στην ουρά. Επίσης με τη προτεινόμενη παράλληλη υλοποίηση, είναι πιθανό πολλές διεργασίες να είναι αδρανείς και επομένως η λύση να μην είναι ιδιαίτερα αποδοτική. Επίσης, η λύση είναι μη πρακτική για μεγάλου μεγέθους

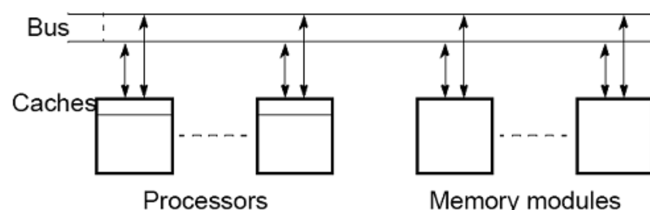
γραφήματα, αν κάθε διεργασία αναλαμβάνει μία μόνο κορυφή. Αυτό μπορεί να λυθεί, αν κάθε διεργασία αναλαμβάνει περισσότερες από μία κορυφές.

Κεφάλαιο 8

Προγραμματισμός σε συστήματα διαμοιραζόμενης μνήμης

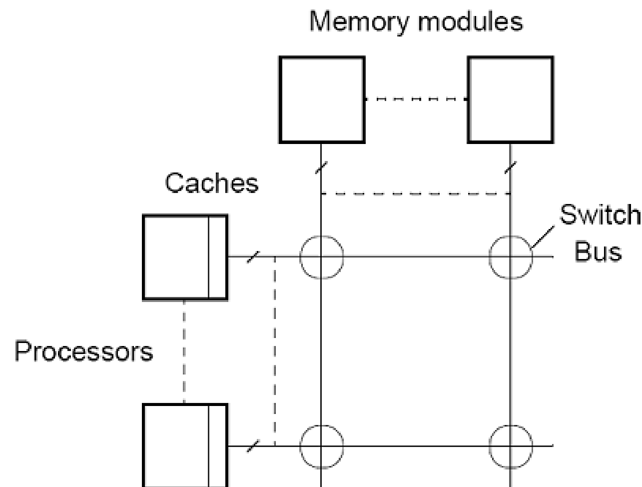
8.1 Πολυεπεξεργαστικά συστήματα διαμοιραζόμενης μνήμης

Σε ένα σύστημα διαμοιραζόμενης μνήμης, κάθε επεξεργαστής μπορεί να προσπελάσει οποιαδήποτε θέση της κοινά διαμοιραζόμενης μνήμης. Ο χώρος διευθύνσεων είναι ίδιος για όλους τους επεξεργαστές, πράγμα που σημαίνει ότι κάθε θέση μνήμης έχει μία μοναδική διεύθυνση. Γενικά, ο προγραμματισμός παράλληλων εφαρμογών σε συστήματα διαμοιραζόμενης μνήμης θεωρείται πιο προσιτός σε σχέση με τον προγραμματισμό σε συστήματα κατανεμημένης μνήμης. Στις περισσότερες περιπτώσεις, η πρόσβαση στα κοινά δεδομένα θα πρέπει να ελέγχεται από τον προγραμματιστή (χρησιμοποιώντας π.χ. κρίσιμές περιοχές). Όταν το πλήθος των επεξεργαστών είναι μικρό, συνήθως χρησιμοποιείται μια αρχιτεκτονική μονού διαύλου, όπου όλοι οι επεξεργαστές και τα τμήματα της μνήμης συνδέονται στον ίδιο δίαυλο. Όταν όμως αυξάνεται το πλήθος των επεξεργαστών, δημιουργείται συμφόρηση στον κοινό δίαυλο. Η ύπαρξη ςαςηε μνήμης σε κάθε επεξεργαστή μειώνει τις προσβάσεις στην κύρια μνήμη. Ακόμα όμως και με αυτή τη βελτίωση, το εύρος ζώνης του διαύλου παραμένει περιορισμένο.



Σχήμα 8.1: Πολυεπεξεργαστικά συστήματα διαμοιραζόμενης μνήμης.

Όταν υπάρχουν πολλοί επεξεργαστές στο σύστημα, άλλα διασυνδετικά δίκτυα προτιμώνται, όπως το crossbar, το οποίο παρέχει πλήρη διασύνδεση μεταξύ των επεξεργαστών και των τμημάτων μνήμης. Μεταξύ οποιουδήποτε επεξεργαστή και μνήμης, υπάρχει μία ξεχωριστή διασύνδεση και έτσι η συμφόρηση αποφεύγεται. Το μειονέκτημα αυτού του δικτύου είναι το υψηλό του κόστος.



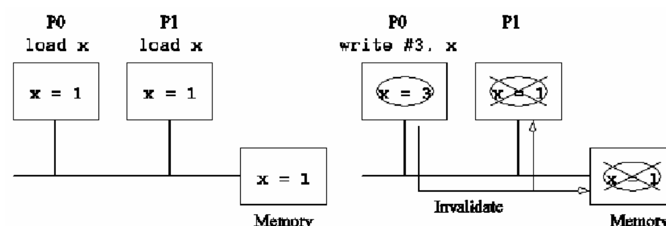
Σχήμα 8.2: Το σύστημα crossbar.

8.2 Διαμοιραζόμενα δεδομένα σε συστήματα με ςαςηε μνήμες

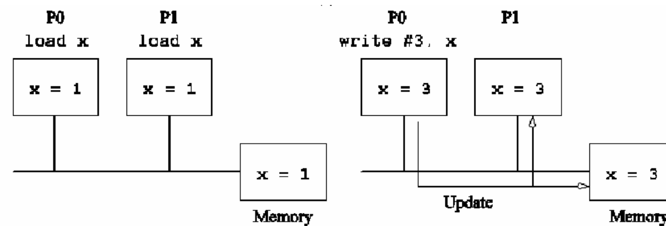
Όπως αναφέρθηκε, σε όλα τα σύγχρονα υπολογιστικά συστήματα, οι επεξεργαστές διαθέτουν μνήμες cache. Οι μνήμες αυτές είναι υψηλών επιδόσεων και χρησιμοποιούνται για να κρατούν δεδομένα στα οποία έγινε προσπέλαση πρόσφατα. Αυτό έχει σαν αποτέλεσμα, πολλαπλά αντίγραφα των περιεχομένων της ίδιας θέσης μνήμης να είναι, πιθανόν, κατανεμημένα στις μνήμες cache διαφορετικών επεξεργαστών. Αν κάποιος από αυτούς τους επεξεργαστές αλλάξει το περιεχόμενο ενός από αυτά τα αντίγραφα, αυτό έχει ως συνέπεια όλα τα αντίγραφα της ίδιας θέσης μνήμης να μην έχουν τα ίδια περιεχόμενα. Σε αυτή την περίπτωση, θα πρέπει να αποκατασταθεί η συνοχή μεταξύ των διαφόρων αντιγράφων.

Υπάρχουν δύο κατηγορίες πρωτοκόλλων συνοχής μνημών cache (cache coherence protocols):

- **Πρωτόκολλα ενημέρωσης**: Τα αντίγραφα σε όλες τις μνήμες cache ενημερώνονται κάθε φορά που ένα αντίγραφο ενημερώνεται.
- **Πρωτόκολλα ακύρωσης**: Όταν ένα αντίγραφο αλλάξει, τα αντίγραφα στις μνήμες cache ακυρώνονται (θέτοντας ένα σχετικό δυαδικό ψηφίο στη μνήμη ςαςηε). Αυτά τα αντίγραφα ενημερώνονται όταν ο σχετικός επεξεργαστής προσπαθήσει να προσπελάσει ένα άκυρο αντίγραφο.



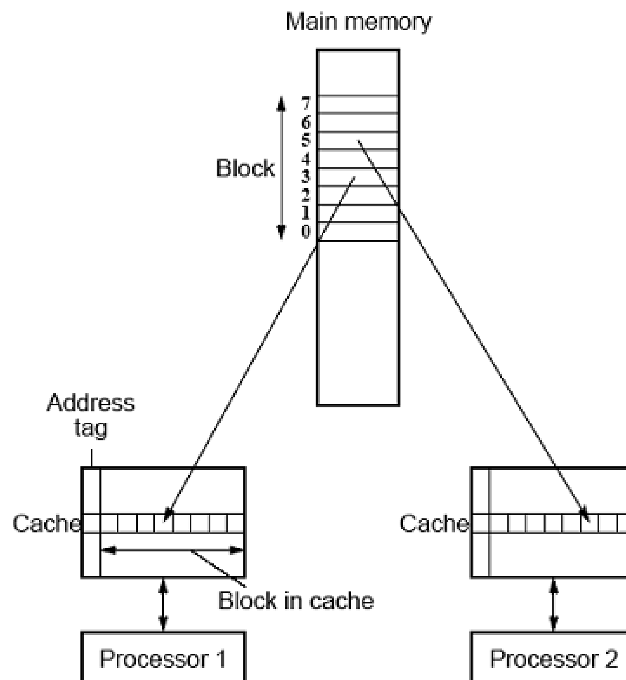
Σχήμα 8.3: Πρωτόκολλο ακύρωσης.



Σχήμα 8.4: Πρωτόκολλο ενημέρωσης.

8.2.1 Ψευδής Διαμοιρασμός (False Sharing)

Οι μνήμες cache δεν αποθηκεύουν μεμονωμένες θέσεις μνήμης αλλά blocks από συνεχόμενες θέσεις μνήμης. Αν ένας επεξεργαστής διαβάσει μία θέση μνήμης, δεν θα μεταφερθεί στη μνήμη cache το περιεχόμενο αυτής της θέσης αλλά όλο το block που περιέχει αυτή τη θέση μνήμης. Αν τώρα διαφορετικοί επεξεργαστές τροποποιούν τα περιεχόμενα διαφορετικών θέσεων μνημών του ίδιου block, θα πρέπει να εκτελεσθεί το πρωτόκολλο συνοχής της μνήμης cache (ενημέρωση ή ακύρωση όλου του block) για κάθε τροποποίηση, παρόλο που οι επεξεργαστές δεν αλλάζουν την ίδια θέση μνήμης.



Σχήμα 8.5: Ψευδής Διαμοιρασμός - Παράδειγμα.

Στο παράδειγμα του Σχήματος 8.5, κάθε block αποτελείται από 8 συνεχόμενες λέξεις. Ο επεξεργαστής 1 διαβάζει τη λέξη 4 του block, ενώ ο επεξεργαστής 2 διαβάζει τη λέξη 6. Η λύση στο πρόβλημα αυτό είναι η τοποθέτηση των δεδομένων στην κύρια μνήμη κατά τέτοιο τρόπο ώστε τα δεδομένα που τροποποιούνται από διαφορετικούς επεξεργαστές να είναι σε διαφορετικά blocks.

8.3 Προγραμματισμός σε Συστήματα Διαμοιραζόμενης Μνήμης

Υπάρχουν πολλές εναλλακτικές για το προγραμματισμό πολυεπεξεργαστών διαμοιραζόμενης μνήμης:

- Χρήση «βαριών» (heavy weight) διεργασιών.
- Χρήση νημάτων π.χ. χρήση της βιβλιοθήκης Pthreads.
- Χρήση συναρτήσεων βιβλιοθηκών που υλοποιούν παράλληλα διάφορους υπολογισμούς μαζί με υπάρχουσες γλώσσες ακολουθιακού προγραμματισμού.
- Χρήση γλωσσών ακολουθιακού προγραμματισμού, συμπληρωμένες με οδηγίες μεταγλωττιστή που περιγράφουν λεπτομέρειες της παράλληλης υλοποίησης. Παράδειγμα είναι το πρότυπο OpenMP.

8.3.1 Χρήση «βαριών» διεργασιών

Τα σύγχρονα λειτουργικά συστήματα βασίζονται στην έννοια της διεργασίας. Σε ένα σύστημα ενός επεξεργαστή, ο χρόνος εκτέλεσης του επεξεργαστή μοιράζεται μεταξύ των διεργασιών, με εναλλαγή εκτέλεσης των διεργασιών. Αυτή η εναλλαγή μπορεί να συμβαίνει ανά τακτά χρονικά διαστήματα ή όταν η διεργασία που έχει τον επεξεργαστή υπό τον έλεγχο της καθυστερήσει για κάποιο λόγο (I/O). Αν και οι διεργασίες μπορούν να χρησιμοποιηθούν στο παράλληλο προγραμματισμό εντούτοις αποφεύγονται λόγω της μεγάλης επιβάρυνσης που έχει ο χειρισμός τους. Πάντως παραλλαγές του βασικού μηχανισμού fork/join για τη δημιουργία διεργασιών έχει υιοθετηθεί από πολλές βιβλιοθήκες παράλληλου προγραμματισμού (π.χ. βιβλιοθήκη Pthreads).

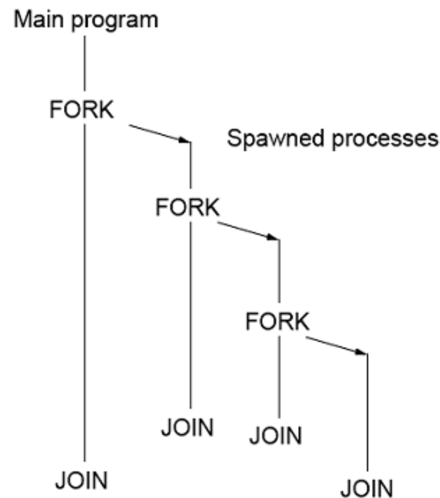
8.3.2 Τεχνική fork/join

Η συνάρτηση συστήματος fork() δημιουργεί μία νέα διεργασία. Η νέα διεργασία (διεργασία παιδί) είναι ακριβές αντίγραφο της διεργασίας που καλεί τη fork (διεργασία πατέρας). Η μόνη διαφορά είναι ότι το αναγνωριστικό της διαδικασίας παιδί είναι διαφορετικό από αυτό της διεργασίας πατέρας. Επίσης έχει ξεχωριστό αντίγραφο των μεταβλητών του πατέρα. Η συνάρτηση fork() επιστρέφει 0 στη διαδικασία παιδί και επιστρέφει το αναγνωριστικό της διαδικασίας παιδί στη διαδικασία πατέρας. Οι διεργασίες «ενώνονται» (joined) χρησιμοποιώντας τις συναρτήσεις συστήματος wait() και exit().

```
    :
pid = fork();                    /* fork */
 Code to be executed by both child and parent
if (pid == 0) exit(0); else wait(0);/* join */
    :
```

Στο παραπάνω παράδειγμα, η διεργασία πατέρας και η διεργασία παιδί εκτελούν το ίδιο τμήμα κώδικα. Ο πατέρας εκτελεί την εντολή wait (pid!=0 στον πατέρα) και περιμένει μέχρι η διαδικασία παιδί να εκτελέσει την εντολή exit (pid=0 στο παιδί).

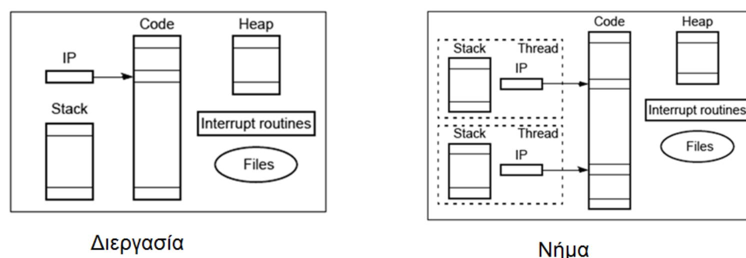
Στο επόμενο τμήμα κώδικα η διαδικασία παιδί εκτελεί διαφορετικό κώδικα από ότι ο πατέρας.



Σχήμα 8.6: Η τεχνική fork/join.

```
pid = fork();
if (pid == 0) {
  code to be executed by slave
} else {
  Code to be executed by parent
}
if (pid == 0) exit(0); else wait(0);
    :
```

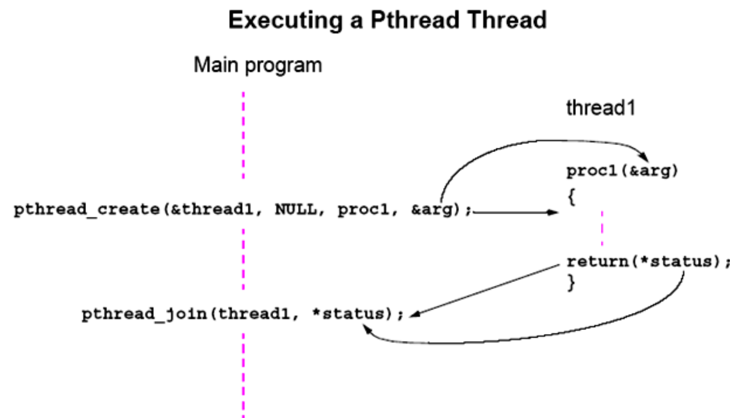
8.3.3 Διεργασίες και νήματα (threads)



Σχήμα 8.7: Διεργασίες και νήματα.

Οι διεργασίες είναι τελείως ξεχωριστά προγράμματα με τις δικές τους μεταβλητές, στοίβα και ιδιωτικό χώρο μνήμης. Η μνήμη μπορεί να διαμοιρασθεί μεταξύ των διεργασιών με τη χρήση συναρτήσεων συστήματος, αλλά γενικά ο χειρισμός των διεργασιών έχει σημαντική επιβάρυνση στο λειτουργικό σύστημα. Πολύ λιγότερη επιβάρυνση έχουμε με τη χρήση νημάτων. Τα νήματα ανήκουν στην ίδια διεργασία και μοιράζονται τον ίδιο κώδικα και σφαιρικές μεταβλητές (heap). Κάθε νήμα έχει ξεχωριστό δείκτη εντολών και επομένως κάθε νήμα έχει ξεχωριστή ροή εκτέλεσης.

Επίσης, κάθε νήμα έχει το δικό του αντίγραφο τοπικών μεταβλητών (stack). Η δημιουργία ενός νήματος είναι πολύ γρηγορότερη σε σχέση με τη δημιουργία μίας διεργασίας Επίσης, ο συγχρονισμός μεταξύ των νημάτων έχει πολύ μικρότερη επιβάρυνση σε σχέση με το συγχρονισμό μεταξύ των διεργασιών.



Σχήμα 8.8: Διεργασίες και νήματα.

Η βιβλιοθηκη Pthreads είναι ένα πρότυπο της IEEE που υλοποιεί βασικές λειτουργίες χειρισμών νημάτων. Στο πρότυπο αυτό, το κύριο πρόγραμμα είναι επίσης νήμα. Ένα νέο νήμα μπορεί να δημιουργηθεί με τη συνάρτηση pthread_create και να καταστραφεί με τη συνάρτηση pthread_join Το νέο νήμα (thread1) εκτελεί την συνάρτηση proc1, της οποίας το όρισμα arg περνάει μέσω της κλήσης της pthread_create.

Μέσω του πρώτου ορίσματος της pthread_create επιστρέφεται στη μεταβλητή thread1 ένα αναγνωριστικό για το νέο νήμα. Αυτό το αναγνωριστικό χρησιμοποιείται στη συνάρτηση pthread_join. Στο δεύτερο όρισμα της pthread_create, περνούν ως είσοδο χαρακτηριστικά που πρέπει να έχει το νέο νήμα. Στο συγκεκριμένο παράδειγμα, το δεύτερο όρισμα έχει τη τιμή NULL που σημαίνει ότι το νέο νήμα θα έχει προκαθορισμένα χαρακτηριστικά. Το κύριο πρόγραμμα εκτελεί την pthread_join και επιστρέφει μόνο όταν το νέο νήμα έχει ολοκληρωθεί. Η τιμή που επιστρέφει η συνάρτηση proc1 που εκτελείται από το νέο νήμα επιστρέφεται τελικά μέσω της pthread_join στο κύριο πρόγραμμα. Αν δεν απαιτείται επιστροφή τιμής από τη διεργασία thread1, το δεύτερο όρισμα της pthread_join τίθεται στην τιμή NULL.

Παράδειγμα: Σύνταξη βασικών εντολών χειρισμού νημάτων

```
#include <pthread.h>
int pthread_create (
    pthread_t *thread_handle, const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);
int pthread_join (
    pthread_t thread,
    void **ptr);
```

Στο προηγούμενο παράδειγμα, μόνο ένα νέο νήμα δημιουργείται. Στο παράδειγμα που ακολουθεί. p νέα νήματα δημιουργούνται. Το κύριο νήμα (κύριο πρόγραμμα) περιμένει την ολοκλήρωση όλων των νημάτων με τη εκτέλεση της συνάρτησης pthread_join p φορές. Παρατηρήστε

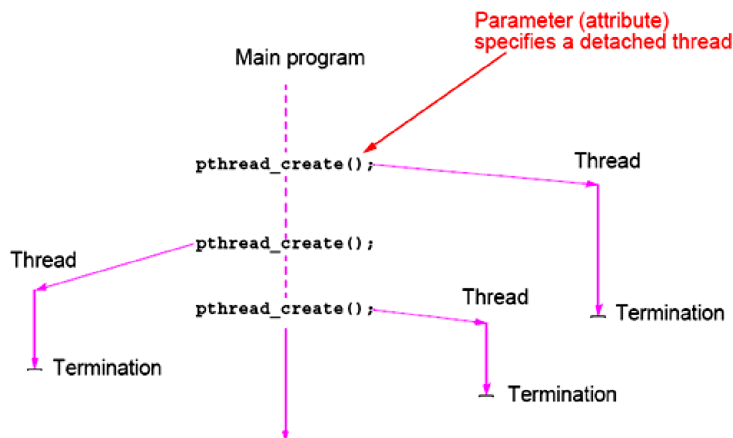
ότι οι πολλαπλές κλήσεις της join λειτουργούν ως φράγμα, αφού το κύριο νήμα δεν προχωράει την εκτέλεσή του περαιτέρω πριν ολοκληρωθούν όλα τα νέα νήματα.

```
for (i=0; i<p; i++)
     pthread_create(&thread[i], NULL, (void * slave), (void *) &arg);

for (i=0; i<p; i++)
     pthread_join(&thread[i], NULL);
```

Αξίζει επίσης να σημειωθεί ότι ένα νήμα μπορεί να μάθει το αναγνωριστικό του με τη κλήση της συνάρτησης pthread_self().

8.3.4 Απομονωμένα (detached) νήματα

Υπάρχει πιθανότητα όπου το κύριο νήμα δεν "ενδιαφέρεται" για την ολοκλήρωση ή μη των υπόλοιπων νημάτων και επομένως δεν χρειάζεται να εκτελέσει τη συνάρτηση pthread_join. Τα νήματα για τα οποία δεν εκτελείται η join από το κύριο νήμα λέγονται απομονωμένα νήματα. Όταν τα απομονωμένα νήματα τερματίζουν, καταστρέφονται από το λειτουργικό σύστημα και οι πόροι του απελευθερώνονται. Ένα νήμα μπορεί να ορισθεί ως απομονωμένο, θέτοντας την κατάλληλη τιμή στο δεύτερο όρισμα της pthread_create(). Τα απομονωμένα νήματα έχουν λιγότερη επιβάρυνση και επομένως θα πρέπει να χρησιμοποιούνται όποτε είναι δυνατόν.



Σχήμα 8.9: Απομονωμένα νήματα.

Παράδειγμα δημιουργίας και τερματισμού νημάτων (υπολογισμός του π):

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
...
    pthread_t p_threads[MAX_THREADS];
```

```
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi, (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
...
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5)
                + (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Αν αντί για την εντολή local_hits ++ είχαμε την εντολή (*hit_pointer)++, θα είχαμε πιο αργή εκτέλεση λόγω του προβλήματος του false sharing. Το πρόβλημα τώρα αποφεύγεται γιατί η μεταβλητή local_hits είναι τοπική μεταβλητή.

8.3.5 Ταυτοχρονισμός - Concurrency

Η εκτέλεση των εντολών μίας διεργασίας ή νήματος εξαρτάται από το λειτουργικό σύστημα. Σε ένα σύστημα μονού επεξεργαστή, τα νήματα/διεργασίες εκτελούνται συνήθως μέχρι να ανασταλεί η εκτέλεσή τους από κάποια λειτουργία, π.χ. I/O λειτουργία. Σε ένα πολυεπεξεργαστικό σύστημα, οι εντολές των νημάτων/διεργασιών επικαλύπτονται χρονικά. Για παράδειγμα, αν υπάρχουν δύο διεργασίες με τις ακόλουθες εντολές

Process 1	Process 2
Instruction 1.1	Instruction 2.1
Instruction 1.2	Instruction 2.2
Instruction 1.3	Instruction 2.3

υπάρχουν πολλές χρονικές επικαλύψεις των εντολών των δύο διεργασιών, π.χ.:

Instruction 1.1
Instruction 1.2
Instruction 2.1
Instruction 1.3
Instruction 2.2
Instruction 2.3

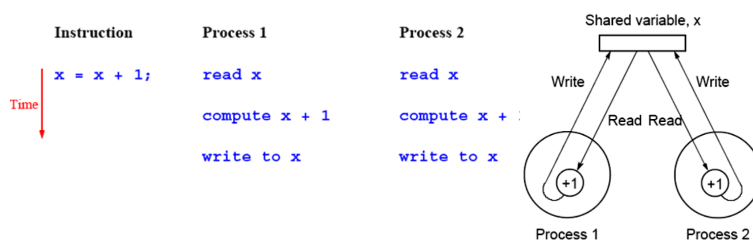
Αν δύο διεργασίες τυπώνουν μηνύματα, τα μηνύματα θα μπορούσαν να εμφανίζονται με διαφορετική σειρά ανάλογα με τη χρονοδρομολόγηση των διεργασιών. Θα μπορούσε να είναι επίσης πιθανό το σενάριο όπου οι χαρακτήρες από διαφορετικά μηνύματα παρεμβάλλονται μεταξύ τους.

8.3.6 Thread-Safe Routines (Ρουτίνες ασφαλείς σε περιβάλλον νημάτων)

Συναρτήσεις συστήματος ή συναρτήσεις βιβλιοθηκών καλούνται thread safe, αν, όταν καλούνται από πολλά νήματα ταυτόχρονα, πάντα παράγουν σωστά αποτελέσματα, π.χ. εκτύπωση μηνυμάτων χωρίς την παρεμβολή χαρακτήρων από άλλα μηνύματα. Όλες οι I/O ρουτίνες είναι συνήθως thread safe και έτσι δεν προκύπτουν προβλήματα με τη παρεμβολή χαρακτήρων. Πάντως, ρουτίνες που προσπελαύνουν διαμοιραζόμενα δεδομένα ή στατικές (static) μεταβλητές μπορεί να μην είναι thread safe. Για παράδειγμα, ρουτίνες που επιστρέφουν χρόνο μπορεί να μην είναι thread safe. Μπορούμε να μετατρέψουμε οποιαδήποτε διαδικασία σε thread safe, αν επιβάλουμε μόνο μία διεργασία να εκτελεί τη διαδικασία. Αυτό μπορεί να επιτευχθεί με την τοποθέτηση της διαδικασίας εντός ενός κρίσιμου τμήματος. Αυτός όμως ο τρόπος επίλυσης του προβλήματος έχει επιπτώσεις στο χρόνο εκτέλεσης.

8.3.7 Προσπέλαση Διαμοιραζόμενων Δεδομένων

Η προσπέλαση διαμοιραζόμενων δεδομένων απαιτεί προσεκτικό χειρισμό. Ας θεωρήσουμε δύο διεργασίες, κάθε μία από τις οποίες προσθέτει τη μονάδα σε μία κοινή μεταβλητή x. Οι ενέργειες που κάνει κάθε διεργασία είναι η ανάγνωση της μεταβλητής x, η πρόσθεση της μονάδας στα περιεχόμενα της x και στη συνέχεια αποθήκευση της νέας τιμής στην μεταβλητή x. Στο σχήμα που ακολουθεί βλέπουμε τη χρονική στιγμή που εκτελούνται οι διάφορες ενέργειες από τις δύο διεργασίες. Αν η αρχική τιμή της x είναι 0, η τελική τιμή της x θα είναι 1 αντί 2, όπως θα ήταν επιθυμητό μετά την εκτέλεση των δύο διεργασιών.



Σχήμα 8.10: Προσπέλαση Διαμοιραζόμενων Δεδομένων.

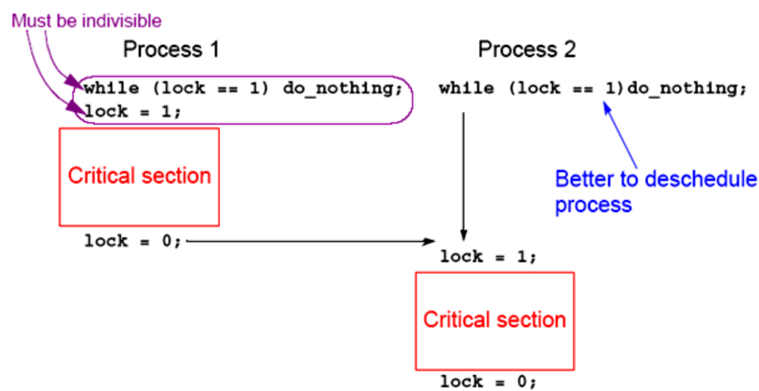
8.3.8 Κρίσιμο Τμήμα (Critical Section)

Στο προηγούμενο παράδειγμα, το πρόβλημα προήλθε από την ταυτόχρονη εκτέλεση του ίδιου τμήματος κώδικα (εντολή $x = x + 1$) από τις δύο διεργασίες. Ένα τμήμα κώδικα το οποίο

πρέπει να εκτελείται από μία το πολύ διεργασία σε οποιαδήποτε χρονική στιγμή ονομάζεται κρίσιμο τμήμα (critical section). Ο μηχανισμός που εξασφαλίζει ότι μόνο μία διεργασία θα εισέρχεται πάντα στο κρίσιμο τμήμα ονομάζεται αμοιβαίος αποκλεισμός. Ο πιο απλός μηχανισμός που εξασφαλίζει αμοιβαίο αποκλεισμό των κρίσιμων τμημάτων είναι τα locks (κλειδαριές). Οι κλειδαριές είναι μεταβλητές boolean, με την τιμή 1 να δηλώνει ότι μία διεργασία έχει εισέλθει στο κρίσιμο τμήμα και με 0 να δηλώνει ότι καμία διεργασία δεν έχει εισέλθει στο κρίσιμο τμήμα.

Οι κλειδαριές δουλεύουν όπως και η κλειδαριά μίας πόρτας:

Μία διεργασία που βρίσκει την "πόρτα" ενός κρίσιμου τμήματος ανοιχτή, εισέρχεται στο κρίσιμο τμήμα και κλειδώνει την πόρτα ώστε να αποτρέψει άλλες διεργασίες να εισέλθουν στο κρίσιμο τμήμα. Από τη στιγμή που η διεργασία ολοκληρώσει την εκτέλεση του κρίσιμου τμήματος, ξεκλειδώνει την πόρτα και φεύγει.



Σχήμα 8.11: Έλεγχος του κρίσιμου τμήματος με βυσψ ωαιτινγ.

Ο πιο απλός τρόπος υλοποίησης των κλειδαριών είναι με τη χρήση ενός βρόχου όπου οι διεργασίες περιμένουν ενόσω η μεταβλητή lock είναι 1 πράγμα που σημαίνει ότι μία άλλη διεργασία είναι ήδη στο κρίσιμο τμήμα. Η διεργασία που εξέρχεται από το κρίσιμο τμήμα θέτει τη μεταβλητή lock ίση με 0 και επομένως κάποια άλλη διεργασία μπορεί να εισέλθει στο κρίσιμο τμήμα. Η διεργασία που περιμένει ελέγχει συνεχώς τη συνθήκη του βρόχου με αποτέλεσμα να δεσμεύει πολύτιμο χρόνο επεξεργασίας (busy waiting). Προτιμότερο θα ήταν η διεργασία να περιμένει σε μία ουρά αναμονής μέχρι να γίνει διαθέσιμο το κρίσιμο τμήμα. Άλλο ένα πρόβλημα που μπορεί να προκύψει είναι να εισέλθουν τελικά δύο διεργασίες στο κρίσιμο τμήμα. Για να αποφευχθεί αυτό θα πρέπει η εντολή while και η επόμενη της να εκτελούνται αδιαίρετα από μία διεργασία χωρίς να παρεμβάλλεται δηλαδή η εκτέλεση κάποιας άλλης διεργασίας.

8.3.9 Λειτουργίες Κλειδώματος του Προτύπου Pthread

Οι κλειδαριές υλοποιούνται στο πρότυπο Pthreads με αμοιβαίως αποκλειόμενες μεταβλητές, γνωστές με το όνομα mutex:

```
...
pthread_mutex_lock(&mutex1);
critical section
pthread_mutex_unlock(&mutex1);
...
```

Αν ένα νήμα φτάσει στην εντολή pthread_mutex_lock και βρει την κλειδαριά mutex1 κλειδωμένη, περιμένει μέχρι η κλειδαριά να ξεκλειδώσει. Αν περισσότερα από ένα νήματα περιμένουν για

να ανοίξει η κλειδαριά, το σύστημα θα επιλέξει ένα νήμα και θα του επιτρέψει να προχωρήσει. Μόνο το νήμα το οποίο έχει κλειδώσει τη μεταβλητή mutex1 μπορεί να την ξεκλειδώσει με την εντολή pthread_mutex_unlock. Ακολουθούν οι βασικές συναρτήσεις χειρισμού των μεταβλητών mutex.

```
int pthread_mutex_lock (pthread_mutex_t *mutex_lock);
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
int pthread_mutex_init (pthread_mutex_t *mutex_lock, const pthread_mutexattr_t *lock_attr);
```

Η εντολή pthread_mutex_init αρχικοποιεί τη μεταβλητή mutex και θα πρέπει να εκτελείται νωρίτερα από τις υπόλοιπες.

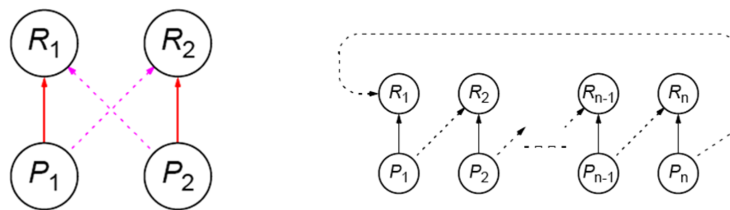
```
#include <pthread.h>
void *find_min(void *list_ptr);
pthread_mutex_t minimum_value_lock;
int minimum_value, partial_list_size;

main() {
/* declare and initialize data structures and list */
minimum_value = MIN_INT;
pthread_init();
pthread_mutex_init(&minimum_value_lock, NULL);

/* initialize lists, list_ptr, and partial_list_size */
/* create and join threads here */
 }

void *find_min(void *list_ptr) {
int *partial_list_pointer, my_min, i;
my_min = MIN_INT;
partial_list_pointer = (int *) list_ptr;

for (i = 0; i < partial_list_size; i++)
    if (partial_list_pointer[i] < my_min)
        my_min = partial_list_pointer[i];
/* lock the mutex associated with minimum_value and update the variable as required */
pthread_mutex_lock(&minimum_value_lock);
if (my_min < minimum_value)
    minimum_value = my_min;
/* and unlock the mutex */

pthread_mutex_unlock(&minimum_value_lock);
pthread_exit(0);
}
```

Στο προηγούμενο πρόγραμμα βρίσκεται το ελάχιστο μίας λίστας. Κάθε νήμα αναλαμβάνει να βρει το ελάχιστο σε ένα τμήμα της αρχικής λίστας και το συνολικό ελάχιστο αποθηκεύεται στη μεταβλητή minimum_value της οποίας ο έλεγχος και τροποποίηση προστατεύονται από τη μεταβλητή mutex minimum_value_lock. Το πρότυπο Pthreads προσφέρει επίσης τη ρουτίνα pthread_mutex_trylock(), η οποία μπορεί να ελέγξει αν μία μεταβλητή mutex είναι κλειδωμένη

χωρίς να μπλοκάρει το νήμα στο σημείο αυτό αν η μεταβλητή είναι κλειδωμένη. Αν η μεταβλητή δεν είναι κλειδωμένη, η trylock κλειδώνει τη μεταβλητή και επιστρέφει 0. Αν η μεταβλητή είναι κλειδωμένη επιστρέφει τη τιμή EBUSY χωρίς να μπλοκάρει τη διαδικασία.

8.3.10 Αδιέξοδο (Deadlock)

Είναι σημαντικό κατά το χειρισμό των κλειδαριών να αποφύγουμε την περίπτωση αδιεξόδου. Αδιέξοδο μπορεί να προκύψει όταν μία διεργασία ζητάει ένα πόρο ο οποίος είναι δεσμευμένος από μία άλλη διεργασία και αυτή η διεργασία ζητάει ένα πόρο ο οποίος κρατείται από τη πρώτη διεργασία.



Σχήμα 8.12: αδιέξοδο.

Στο σχήμα αριστερά, έχουμε ένα αδιέξοδο που εμπλέκει δύο διεργασίες.

Το αδιέξοδο μπορεί να συμβεί και σε κυκλική διάταξη όπου κάθε διεργασία κατέχει έναν πόρο ο οποίος ζητείται από μία άλλη διεργασία.

8.3.11 Σημαφόροι (Semaphores)

Αμοιβαίο αποκλεισμό μπορούμε να πετύχουμε με το μηχανισμό των σημαφόρων. Ο σημαφόρος s είναι ένας θετικός ακέραιος (συμπεριλαμβανομένου του μηδενός), στον οποίο ορίζονται οι ακόλουθες δύο λειτουργίες:

- Λειτουργία P: Η λειτουργία αυτή περιμένει μέχρι ο s να γίνει μεγαλύτερος του μηδενός και στη συνέχεια μειώνει τον s κατά 1 και επιτρέπει τη διαδικασία να συνεχίσει. Οι λειτουργίες που περιμένουν τοποθετούνται σε μία ουρά αναμονής και μία από αυτές επιλέγεται όταν εκτελεσθεί η λειτουργία V.

- Λειτουργία V: Η λειτουργία αυτή αυξάνει τον s κατά 1 και ελευθερώνει μία από τις διεργασίες που περιμένουν (αν υπάρχουν). Αμοιβαίος αποκλεισμός μπορεί να επιτευχθεί με ένα σημαφόρο ο οποίος έχει μόνο δύο τιμές, 0 ή 1 (δυαδικός σημαφόρος). Στο παράδειγμα που ακολουθεί, ο σημαφόρος s αρχικοποιείται στη τιμή 1 και στη συνέχεια μόνο μία διεργασία επιτρέπεται να εισέλθει στη κρίσιμη περιοχή.

Process 1	Process 2	Process 3
Noncritical section	Noncritical section	Noncritical section
......	....	....
P(s)	P(s)	P(s)
Critical section	Critical section	Critical section
V(s) V(s) V(s)	....	....
Noncritical section	Noncritical section	Noncritical section

Στη γενική περίπτωση, ένας σημαφόρος μπορεί να πάρει οποιαδήποτε τιμή πέρα από τις δυαδικές τιμές. Παρέχει ένα μέσο για να καταγράψουμε το πλήθος των πόρων που είναι διαθέσιμα προς χρήση και βρίσκουν εφαρμογή στα προβλήματα παραγωγών/καταναλωτών (producer/consumer problems).

Αν και το πρότυπο Pthreads δεν παρέχει σημαφόρους, μπορούμε να χρησιμοποιήσουμε τους σημαφόρους που περιέχει το POSIX Unix. Οι βασικές λειτουργίες χειρισμού των σημαφόρων είναι οι ακόλουθες:

```
#include <semaphore.h>
int sem init( sem t*  semaphore p /* out */, int shared /* in */,
    unsigned initial val /* in */);
int sem destroy(sem t* semaphore p /* in/out  */);
int sem post(sem t* semaphore p / in/out  */);
int sem wait(sem t* semaphore p /* in/out */);
```

Η λειτουργία sem_init αρχικοποιεί το σημαφόρο σε μία αρχική τιμή, η sem_destroy καταστρέφει το σημαφόρο, η sem_post είναι η λειτουργία *V* και η sem_wait είναι η λειτουργία *P*.

Στο παράδειγμα αυτό, τα νήματα είναι σε διάταξη δακτυλίου και κάθε νήμα στέλνει στο επόμενο του στη διάταξη. Κάθε νήμα περιμένει στο δικό του σημαφόρο μέχρι το προηγούμενο από αυτό νήμα να τον ξεκλειδώσει.

```
/ messages is allocated and initialized to NULL in main /
/ semaphores is allocated and initialized to 0 (locked) in
main /
 void Send msg(void rank) {
 long my rank = (long) rank;
 long dest = (my rank + 1) % thread count;
 char my msg = malloc(MSG MAXsizeof(char));

 sprintf(my msg, "Hello to %ld from %ld", dest, my rank);
 messages[dest] = my msg;
 sem post(&semaphores[dest])
/ ````Unlock'''' the semaphore of dest /

 / Wait for our semaphore to be unlocked /
 sem wait(&semaphores[my rank]);
 printf("Thread %ld > %snn", my rank, messages[my rank]);

return NULL;
} / Send msg /
```

8.3.12 Μεταβλητές Συνθήκης (Condition Variables)

Συχνά, ένα κρίσιμο τμήμα εκτελείται μόνο όταν μία συγκεκριμένη συνθήκη ικανοποιηθεί, π.χ. όταν μία μεταβλητή πάρει μία συγκεκριμένη τιμή. Με τη χρήση κλειδαριών, η μεταβλητή θα πρέπει να ελέγχεται ανά τακτά χρονικά διαστήματα, με αποτέλεσμα τη σπατάλη πολύτιμων υπολογιστικών πόρων. Το πρόβλημα αυτό μπορεί να αποφευχθεί με τη χρήση των μεταβλητών συνθήκης (condition variables). Ουσιαστικά υλοποιείται η ακόλουθη λειτουργικότητα:

```
lock mutex;
if condition has occurred
    signal thread(s);
else { unlock the mutex and block;
 }
unlock mutex; /* when thread is unblocked, mutex is relocked */
```

Οι βασικές ρουτίνες χειρισμού των μεταβλητών συνθήκης είναι οι ακόλουθες:

```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *cond,
const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Η pthread_cond_init και η pthread_cond_destroy δημιουργεί και καταστρέφει τη μεταβλητή συνθήκης αντίστοιχα.
- Η pthread_cond_signal θα ξεμπλοκάρει ένα νήμα που περιμένει, ενώ pthread_cond_broadcast ξεμπλοκάρει όλες τα νήματα που περιμένουν.
- Η pthread_cond_wait θα ξεκλειδώσει τη μεταβλητή mutex και στη συνέχεια το νήμα θα μπλοκάρει μέχρι να ξεμπλοκάρει από τη κλήση της συνάρτησης pthread_cond_signal ή της συνάρτησης pthread_cond_broadcast. Ουσιαστικά κατά τη κλήση της pthread_cond_wait εκτελούνται αδιαίρετα τα εξής:

```
pthread mutex unlock(&mutex p);
wait on signal(&cond var p);
pthread mutex lock(&mutex p);
```

Πάντα μία μεταβλητή συνθήκης θα πρέπει να συνδυάζεται με μία μεταβλητή κλειδώματος.
Ένα παράδειγμα χρήσης των μεταβλητών συνθήκης ακολουθεί:

```
pthread_cond_t cond1;
pthread_mutex_t mutex1;
.....
pthread_cond_init(&cond1, NULL);
pthread_mutex_init (&mutex1, NULL);

action()
{
.....
pthread_mutex_lock(&mutex1);
while (c !=0)
  pthread_cond_wait(cond1,mutex1);
pthread_mutex_unlock(&mutex1);
take_action();
```

```
....
}

counter()
{
.....
pthread_mutex_lock(&mutex1);
c--;
if (c==0) pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);
....
}
```

Στο προηγούμενο παράδειγμα, ένα ή περισσότερα νήματα αναλαμβάνουν δράση όταν ο μετρητής *c* γίνει μηδέν. Ένα άλλο νήμα είναι υπεύθυνο για τη μείωση αυτού του μετρητή. Επίσης γίνεται η υπόθεση ότι η ρουτίνα action φτάνει στο κρίσιμο τμήμα πρώτα, αφού τα σήματα (signals) δεν αποθηκεύονται κάπου και μπορούν να χαθούν αν η ρουτίνα counter φτάσει πρώτη στο κρίσιμο τμήμα.

Ο κώδικας που ακολουθεί υλοποιεί ένα φράγμα με τη βοήθεια των μεταβλητών συνθήκης. Συγκεκριμένα, όλα τα νήματα πρέπει να περιμένουν μέχρι thread_count νήματα φθάσουν στο φράγμα.

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void Thread_work(. . .) { . . .
/* Barrier */
pthread_mutex_lock(&mutex);
counter++;
if (counter == thread_count) {
counter = 0;
pthread_cond_broadcast(&cond_var);
} else {
 while (pthread_cond_wait(&cond_var, &mutex) != 0);
}
pthread_mutex_unlock(&mutex);
. . .
}
```

8.3.13 OpenMP

Η βιβλιοθήκη OpenMP είναι ένα πρότυπο για το Παράλληλο Προγραμματισμό. Είναι μία εναλλακτική πρόταση σε σχέση με τις βιβλιοθήκες παράλληλου προγραμματισμού που έχουμε μελετήσει (MPI και Pthreads). Στην OpenMP, ξεκινούμε από μια γλώσσα ακολουθιακού προγραμματισμού και μετατρέπουμε τμήματα του κώδικα σε παράλληλο κώδικα εισάγοντας στο πρόγραμμα συγκεκριμένες οδηγίες (directives) μεταγλωττιστή. Οι οδηγίες αυτές προσδιορίζουν τον τρόπο παραλληλοποίησης του σχετικού κώδικα, παρέχουν λεπτομέρειες για το συγχρονισμό

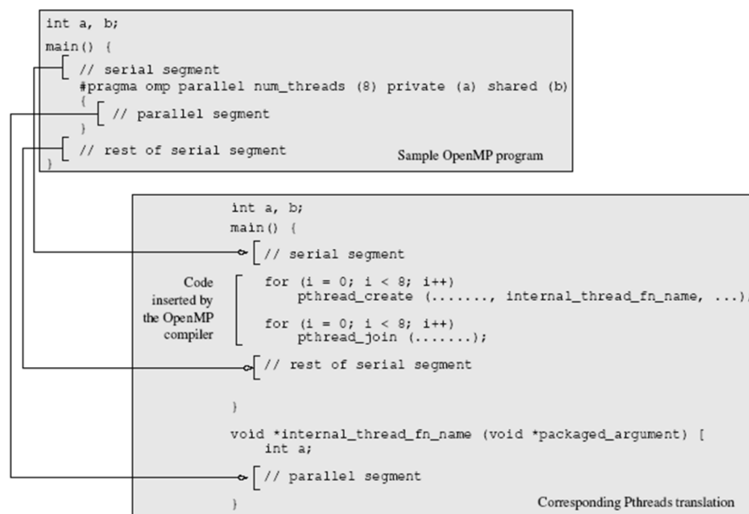
μεταξύ των νημάτων, τον χειρισμό των δεδομένων, χωρίς να χρειάζεται ο προγραμματιστής να χρησιμοποιήσει μεταβλητές κλειδωματος, μεταβλητές συνθήκης κτλ. Οι οδηγίες OpenMP στη C και C++ βασίζονται στην οδηγία #pragma του μεταγλωττιστή. Μία οδηγία αποτελείται από το όνομα της οδηγίας και ακολουθείται από μία λίστα προτάσεων.

```
#pragma omp directive [clause list]
```

Τα προγράμματα σε OpenMP εκτελούνται ακολουθιακά μέχρι να συναντήσουν την οδηγία parallel, η οποία δημιουργεί μία ομάδα νημάτων:

```
#pragma omp parallel [clause list]
/* structured block */
```

Όταν το κύριο νήμα συναντήσει την οδηγία parallel, γίνεται ο master αυτής της ομάδας των νημάτων και το αναγνωριστικό αυτού του νήματος είναι 0 μέσα σε αυτή την ομάδα.



Σχήμα 8.13: Αντιστοιχία των εντολών της OpenMP και των εντολών του Pthreads.

Η λίστα των προτάσεων προσδιορίζει τον υπό συνθήκη παραλληλισμό, το πλήθος των νημάτων ή τον χειρισμό των δεδομένων.

- **Υπό συνθήκη παραλληλισμός**: Η συνθήκη if (scalar expression) προσδιορίζει αν η οδηγία parallel θα έχει ως αποτέλεσμα τη δημιουργία νημάτων.
- **Βαθμός παραλληλισμού**: Η πρόταση num_threads (integer expression) προσδιορίζει το πλήθος των νημάτων που θα δημιουργηθούν.
- **Χειρισμός δεδομένων**: Η πρόταση private (variable list) προσδιορίζει ποιες μεταβλητές θα είναι τοπικές σε κάθε νήμα. Η πρόταση shared (variable list) προσδιορίζει ποιες μεταβλητές θα είναι κοινές σε όλες τις διεργασίες.

Για παράδειγμα:

```
#pragma omp parallel if (is_parallel== 1) num_threads(8)
private (a) shared (b) {
 /* structured block */
}
```

Αν η τιμή της μεταβλητής is_parallel είναι ίση με 1, οκτώ νήματα δημιουργούνται. Η μεταβλητή a είναι ιδιωτική σε κάθε νήμα ενώ η μεταβλητή b είναι κοινή σε όλα τα νήματα. Η πρόταση reduction προσδιορίζει πώς πολλαπλά τοπικά αντίγραφα μίας μεταβλητής σε διαφορετικά νήματα συνδυάζονται σε ένα μοναδικό αντίγραφο στο νήμα master όταν όλα τα νήματα τερματίζουν. Η πρόταση reduction συντάσσεται ως εξής: reduction (operator: variable list). Οι μεταβλητές στη λίστα ορίζονται αυτόματα ως ιδιωτικές σε κάθε νήμα. Ο τελεστής operator μπορεί να είναι ένας από τους ακόλουθους τελεστές:

`+, *, -, &, |, ^, &&, ||.`

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

Ο παρακάτω κώδικας υπολογίζει την τιμή του π με τη χρήση της οδηγίας reduction:

```
/* *****************************************************
An OpenMP version of a threaded program to compute PI.
***************************************************** */
#pragma omp parallel default(private) shared (npoints)
reduction(+: sum) num_threads(8)
{
num_threads = omp_get_num_threads();
sample_points_per_thread = npoints / num_threads;
sum = 0;
for (i = 0; i < sample_points_per_thread; i++) {
    rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
    rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
        (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
        sum ++;
}
}
```

Η οδηγία parallel μπορεί να χρησιμοποιηθεί μαζί με τις οδηγίες for και sections για να προσδιορίζουμε πώς θα παραλληλοποιηθεί ένας επαναληπτικός βρόχος ή πώς θα εκτελεστούν παράλληλα διαφορετικά τμήματα κώδικα.

Η οδηγία for χρησιμοποιείται για να μοιράσουμε τις επαναλήψεις ενός βρόχου στα διαθέσιμα νήματα. Η γενική μορφή για την οδηγία for έχει ως εξής:

```
#pragma omp for [clause list]
/* for loop */
```

Με τη χρήση της οδηγίας for, ο υπολογισμός του π μπορεί να απλοποιηθεί τώρα ως εξής:

```
#pragma omp parallel default(private) shared (npoints)
reduction(+: sum) num_threads(8)
{
sum = 0;
```

```
#pragma omp for
for (i = 0; i < npoints; i++) {
    rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
    rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
        (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
        sum ++;
}
}
```

Η πρόταση schedule της οδηγίας for καθορίζει τον τρόπο που θα μοιραστούν οι επαναλήψεις στα νήματα. Η γενική μορφή της οδηγίας schedule είναι schedule(scheduling_class[, chunksize]). Η OpenMP υποστηρίζει τέσσερις κλάσεις χρονοδρομολόγησης: static, dynamic, guided kai runtime.

```
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
reduction(+:sum) schedule(static,1)
for (i = 0; i <= n; i++)
    sum += f(i);
```

Στο παράδειγμα, αν υπάρχουν τρία νήματα (thread_count=3) και ($n = 9$) επαναλήψεις, τα νήματα θα αναλάβουν τις εξής επαναλήψεις:

- Νήμα 1: 1,4,7
- Νήμα 2: 2, 5, 8
- Νήμα 3: 3, 6, 9

Αν είχαμε χρησιμοποιήσει την οδηγία schedule(static,2), τα νήματα θα αναλάβουν τις εξής επαναλήψεις:

- Νήμα 1: 1,2, 7, 8
- Νήμα 2: 3, 4, 9
- Νήμα 3: 5, 6

Γενικά, στη δρομολόγηση static, οι επαναλήψεις κατανέμονται στις διεργασίες εκ των προτέρων πριν ο βρόχος εκτελεστεί. Στις δρομολογήσεις dynamic και guided, οι επαναλήψεις κατανέμονται στα νήματα δυναμικά καθώς ο βρόχος εκτελείται. Όταν ένα νήμα ολοκληρώνει ένα σύνολο επαναλήψεων, μπορεί να ζητήσει νέες επαναλήψεις από το σύστημα. Η δρομολόγηση runtime καθορίζεται την ώρα της εκτέλεσης ενώ στην auto, η δρομολόγηση καθορίζεται είτε στη φάση της μεταγλώττισης ή την ώρα της εκτέλεσης.

Πέρα από τη παράλληλη εκτέλεση των επαναλήψεων ενός βρόχου, η OpenMP υποστηρίζει και την παράλληλη υλοποίηση οποιωνδήποτε τμημάτων κώδικα με τη χρήση της οδηγίας sections. Η γενική μορφή της οδηγίας sections έχει ως εξής:

```
#pragma omp sections [clause list]
{
[#pragma omp section
```

```
/* structured block */
]
[#pragma omp section
/* structured block */
]
...
}
```

Στο παράδειγμα που ακολουθεί, δημιουργούνται τρία νήματα τα οποία εκτελούν παράλληλα τα έργα A, B και C.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

8.3.14 Οδηγίες συγχρονισμού

Η OpenMP προσφέρει διάφορες οδηγίες για το συγχρονισμό των νημάτων Η οδηγία critical επιτρέπει σε ένα μόνο νήμα να εισέλθει στο σχετικό block κώδικα. Συγκεκριμένα, όταν ένα ή περισσότερα νήματα φτάσουν στην οδηγία critical:

```
#pragma omp critical name
structured_block
```

περιμένουν μέχρι κανένα άλλο νήμα να μην εκτελεί το ίδιο κρίσιμο τμήμα και τότε μόνο ένα από αυτά τα νήματα προχωρεί για να εκτελέσει το κρίσιμο τμήμα. Στον κώδικα που ακολουθεί, η ουρά θεωρείται κρίσιμο τμήμα και η πρόσβαση σε αυτή προστατεύεται με τις οδηγίες critical.

```
#pragma omp parallel sections
 {
 #pragma parallel section
 {
 /* producer thread */
 task = produce_task();
 #pragma omp critical ( task_queue)
```

```
 {
 insert_into_queue(task);
 }}
 #pragma parallel section
 {
 /* consumer thread */
 #pragma omp critical ( task_queue)
{
task = extract_from_queue(task);
}
consume_task(task);
}}
```

Επίσης, η OpenMP προσφέρει και το μηχανισμό του φράγματος που υλοποιείται με την οδηγία barrier

```
#pragma omp barrier
```

Με την εισαγωγή αυτής της οδηγίας στο κώδικα, τα νήματα περιμένουν να φτάσουν όλα μαζί στο φράγμα πριν προχωρήσουν.

Η οδηγία atomic

```
#pragma omp atomic
expression_statement
```

υλοποιεί ένα κρίσιμο τμήμα αποδοτικά όταν το κρίσιμο τμήμα είναι μία απλή εντολή (πρόσθεση, αφαίρεση ή άλλη αριθμητική εντολή σε μία μεταβλητή η οποία ορίζεται από την expression_statement).

8.4 Παραδείγματα προγραμμάτων διαμοιραζόμενης μνήμης

Άθροιση των στοιχείων ενός πίνακα 1000 στοιχείων, $a[1000]$:

```
int sum, a[1000];
sum = 0;
for (i = 0; i < 1000; i++)
sum = sum + a[i];
```

Διεργασίες UNIX Ο Υπολογισμός διαιρείται σε δύο τμήματα. Το ένα τμήμα θα αθροίζει τις ζυγές θέσεις και το άλλο τις μονές, δηλαδή:

```
Process 1:
sum1 = 0;
for (i = 0; i < 1000; i = i + 2)
    sum1 = sum1 + a[i];

Process 2:
    sum2 = 0;
    for (i = 1; i < 1000; i = i + 2)
        sum2 = sum2 + a[i];
```

Κάθε διεργασία θα προσθέσει το αποτέλεσμά της στη μεταβλητή sum:

```
sum = sum + sum1;                    sum = sum + sum2;
```

Η μεταβλητή sum επειδή είναι κοινή μεταβλητή θα πρέπει να προστατευθεί με μηχανισμό κλειδώματος. Επίσης και ο πίνακας a είναι κοινός στις δύο διεργασίες:

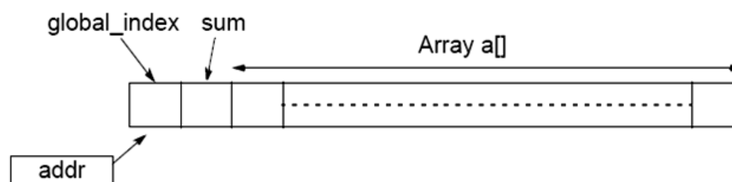
8.4.1 Θέσεις μνήμης με πρόσβαση και από τις δύο διεργασίες UNIX

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>
#define array_size 1000        /* no of elements in shared memory */
extern char *shmat();
void P(int *s);
void V(int *s);
int main()
{
int shmid, s, pid;             /* shared memory, semaphore, proc id */
char *shm;                     /*shared mem. addr returned by shmat()*/
int *a, *addr, *sum;           /* shared data variables*/
int partial_sum;               /* partial sum of each process */
int i;
                               /* initialize semaphore set */
int init_sem_value = 1;
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT))
if (s == -1) {                 /* if unsuccessful*/
  perror("semget");
  exit(1);
}
if (semctl(s, 0, SETVAL, init_sem_value) < 0) {
  perror("semctl");
  exit(1);
}



                                 /* create segment*/
shmid = shmget(IPC_PRIVATE,(array_size*sizeof(int)+1),
   (IPC_CREAT|0600));
if (shmid == -1) {
  perror("shmget");
  exit(1);
}
                               /* map segment to process data space */
shm = shmat(shmid, NULL, 0);
                               /* returns address as a character*/
if (shm == (char*)-1) {
  perror("shmat");
  exit(1);
}
```

```
addr = (int*)shm;              /* starting address */
sum = addr;                    /* accumulating sum */
addr++;
a = addr;                      /* array of numbers, a[] */

*sum = 0;
for (i = 0; i < array_size; i++) /* load array with numbers */
  *(a + i) = i+1;

pid = fork();                  /* create child process */
if (pid == 0)    {             /* child does this */
  partial_sum = 0;
  for (i = 0; i < array_size; i = i + 2)
    partial_sum += *(a + i);
else {                         /* parent does this */
  partial_sum = 0;
  for (i = 1; i < array_size; i = i + 2)
    partial_sum += *(a + i);
}
P(&s);                         /* for each process, add partial sum */
  *sum += partial_sum;
V(&s);



printf("\nprocess pid = %d, partial sum = %d\n", pid, partial_sum);
if (pid == 0) exit(0); else wait(0);      /* terminate child proc */
printf("\nThe sum of 1 to %i is %d\n", array_size, *sum);

                                          /* remove semaphore */
if (semctl(s, 0, IPC_RMID, 1) == -1) {
  perror("semctl");
  exit(1);
}
                                          /* remove shared memory */
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
  perror("shmctl");
  exit(1);
}
}                                         /* end of main */



void P(int *s)                             /* P(s) routine*/
{
  struct sembuf sembuffer, *sops;
  sops = &sembuffer;
  sops->sem_num = 0;
  sops->sem_op = -1;
  sops->sem_flg = 0;
  if (semop(*s, sops, 1) < 0) {
    perror("semop");
    exit(1);
  }
return;
}

void V(int *s)                             /* V(s) routine */
{
  struct sembuf sembuffer, *sops;
  sops = &sembuffer;
  sops->sem_num = 0;
  sops->sem_op = 1;
  sops->sem_flg = 0;
  if (semop(*s, sops, 1) <0) {
    perror("semop");
    exit(1);
  }
return;
}
```

```
SAMPLE OUTPUT

process pid = 0, partial sum = 250000
process pid = 26127, partial sum = 250500
The sum of 1 to 1000 is 500500
```

Παράδειγμα με Pthreads: Δημιουργούνται *n* νήματα και κάθε νήμα παίρνει αριθμούς από τη λίστα και τους αθροίζει. Όταν ολοκληρωθούν οι τοπικοί υπολογισμοί, τα νήματα προσθέτουν τα μερικά τους αποτελέσματα σε μία κοινά διαμοιραζόμενη μεταβλητή sum. Επίσης χρησιμοποιείται η κοινά διαμοιραζόμενη μεταβλητή global_index. Μετά από κάθε ανάγνωση της global_index, η μεταβλητή αυτή αυξάνεται για να δείχνει το επόμενο προς επεξεργασία στοιχείο του πίνακα *a*[]. Το τελικό αποτέλεσμα θα αποθηκευθεί στη μεταβλητή sum, η οποία όπως και προηγουμένως είναι κοινή σε όλα τα νήματα και επομένως πρέπει προστατεύεται από μηχανισμό κλειδώματος.




```
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define no_threads 10
                              /* shared data */
int a[array_size];            /* array of numbers to sum */
int global_index = 0;         /* global index */
int sum = 0;                  /* final result, also used by slaves */
pthread_mutex_t mutex1;       /* mutually exclusive lock variable */
void *slave(void *ignored)    /* Slave threads */
{
int local_index, partial_sum = 0;
do {
  pthread_mutex_lock(&mutex1);/* get next index into the array */
    local_index = global_index;/* read current index & save locally*/
    global_index++;           /* increment global index */
  pthread_mutex_unlock(&mutex1);

  if (local_index < array_size) partial_sum += *(a + local_index);
} while (local_index < array_size);

pthread_mutex_lock(&mutex1);  /* add partial sum to global sum */
  sum += partial_sum;
pthread_mutex_unlock(&mutex1);

return ();                    /* Thread exits */
}
```

```
main () {
int i;
pthread_t thread[10];                   /* threads */
pthread_mutex_init(&mutex1,NULL);       /* initialize mutex */

for (i = 0; i < array_size; i++)        /* initialize a[] */
  a[i] = i+1;

for (i = 0; i < no_threads; i++)        /* create threads */
  if (pthread_create(&thread[i], NULL, slave, NULL) != 0)
    perror("Pthread_create fails");

for (i = 0; i < no_threads; i++)        /* join threads */
  if (pthread_join(thread[i], NULL) != 0)
    perror("Pthread_join fails");
printf("The sum of 1 to %i is %d\n", array_size, sum);
}                                       /* end of main */

SAMPLE OUTPUT

The sum of 1 to 1000 is 500500
```

Παράδειγμα σε Java:

```
public class Adder
{
  public int[] array;
  private int sum = 0;
  private int index = 0;
  private int number_of_threads = 10;
  private int threads_quit;

  public Adder()
  {
    threads_quit = 0;
    array = new int[1000];
    initializeArray();
    startThreads();
  }

  public synchronized int getNextIndex()
  {
    if(index < 1000) return(index++); else return(-1);
  }
```

```
  public synchronized void addPartialSum(int partial_sum)
  {
    sum = sum + partial_sum;
    if(++threads_quit == number_of_threads)
      System.out.println("The sum of the numbers is " + sum);
  }

  private void initializeArray()
  {
    int i;
    for(i = 0;i < 1000;i++) array[i] = i;
  }

  public void startThreads()
  {
    int i = 0;
    for(i = 0;i < 10;i++)
    {
      AdderThread at = new AdderThread(this,i);
      at.start();
    }
}
  {
    Adder a = new Adder();
  }

}


    public static void main(String args[])

  class AdderThread extends Thread
  {
    int partial_sum = 0;
    Adder parent;
    int number;
    public AdderThread(Adder parent,int number)
    {
      this.parent = parent;
      this.number = number;
    }

    public void run()
    {
      int index = 0;
      while(index != -1) {
        partial_sum = partial_sum + parent.array[index];
        index = parent.getNextIndex();
      }
      System.out.println("Partial sum from thread " + number + " is "
          + partial_sum);
      parent.addPartialSum(partial_sum);
    }
  }
```

Κεφάλαιο 9

Αλγόριθμοι ταξινόμησης

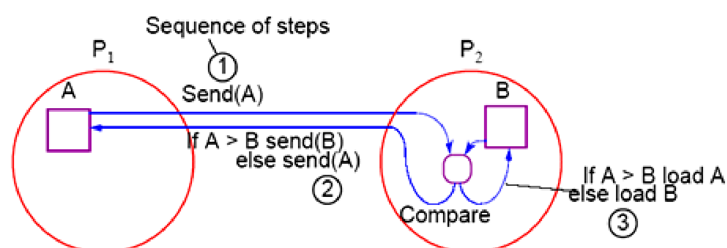
Το πρόβλημα της ταξινόμησης είναι η επαναδιάταξη των αριθμών μίας λίστας σε αύξουσα (ή φθίνουσα) διάταξη. Οι βέλτιστοι ακολουθιακοί αλγόριθμοι οι οποίοι δεν χρησιμοποιούν ειδικές ιδιότητες των στοιχείων εισόδου έχουν πολυπλοκότητα χειρότερης περίπτωση $O(n \log n)$, όπου n είναι το πλήθος των στοιχείων που ταξινομούνται. Επομένως, η βέλτιστη πολυπλοκότητα ενός παράλληλου αλγόριθμου ταξινόμησης με n διεργασίες θα είναι $O(n \log n)/n = O(\log n)$. Πράγματι, υπάρχει παράλληλος αλγόριθμος που επιτυγχάνει αυτή την πολυπλοκότητα, αλλά η σταθερά που κρύβεται στον ασυμπτωτικό συμβολισμό είναι πολύ μεγάλη. Γενικά, ένας πρακτικός $O(\log n)$ αλγόριθμος με n επεξεργαστές είναι ένας στόχος ο οποίος δεν είναι εύκολο να επιτευχθεί με αλγόριθμους ταξινόμησης βασιζόμενοι μόνο σε συγκρίσεις (comparison-based sorting algorithms).

Η λειτουργία compare-and-exchange αποτελεί τη βάση των περισσότερων ακολουθιακών αλγόριθμων ταξινόμησης. Σε αυτή τη λειτουργία, δύο αριθμητικές μεταβλητές, A και B, συγκρίνονται. Αν $A > B$, τότε οι A, B εναλλάσσονται, δηλαδή

```
if (A > B) {
    temp = A;
    A = B;
    B = temp;
}
```

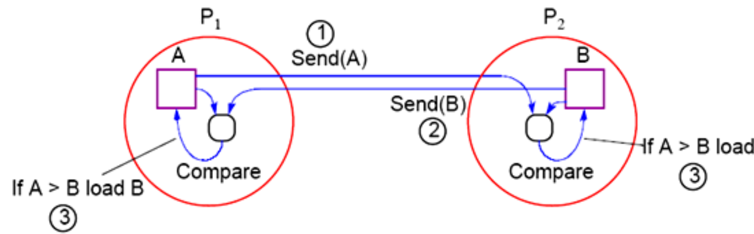
Η λειτουργία compare-and-exchange ταιριάζει σε ένα σύστημα κατανεμημένης μνήμης.

Στο σχήμα που ακολουθεί, απεικονίζεται ο πρώτος τρόπος υλοποίησης της compare-and-exchange, με δύο διεργασίες οι οποίες μπορούν να ανταλλάξουν μηνύματα.



Η $P1$ στέλνει το περιεχόμενο της A στη διεργασία $P2$. Η $P2$ συγκρίνει την A και τη B και στέλνει το περιεχόμενο της B στη διεργασία $P1$ αν η A είναι μεγαλύτερη της B (διαφορετικά επιστρέφει ξανά το περιεχόμενο της A στη $P1$).

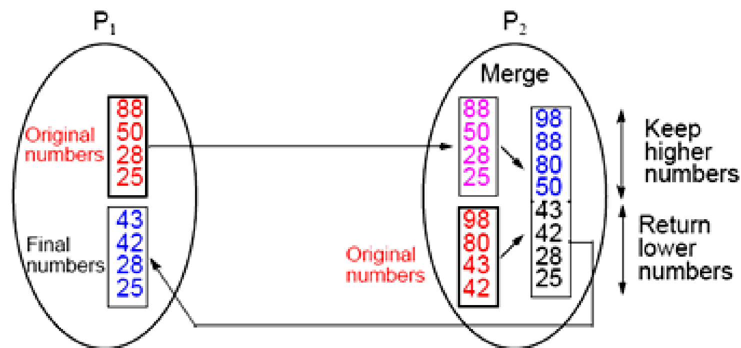
Στο επόμενο σχήμα έχουμε μία εναλλακτική υλοποίηση:



Η $P1$ στέλνει το περιεχόμενο της μεταβλητής A στη διεργασία $P2$ και η διεργασία $P2$ στέλνει το περιεχόμενο της μεταβλητής B στη διεργασία $P1$. Στη συνέχεια, και οι δύο διεργασίες εκτελούν τις λειτουργίες σύγκρισης. Η $P1$ κρατάει το μικρότερο από τους δύο αριθμούς A και B και η $P2$ κρατάει το μεγαλύτερο των αριθμών A και B.

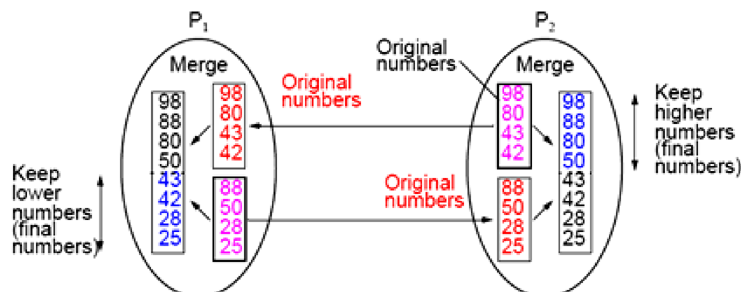
9.1 Χωρισμός Δεδομένων (Data Partitioning)

Συνήθως το πλήθος n των στοιχείων που θα ταξινομηθούν είναι πολύ μεγαλύτερο από το πλήθος p των επεξεργαστών/διεργασιών. Σε αυτή την περίπτωση κάθε διεργασία αναλαμβάνει n/p στοιχεία. Η λειτουργία compare-and-exchange γενικεύεται σε αυτή την περίπτωση, όπως φαίνεται στο επόμενο σχήμα.



Η διεργασία $P1$ στέλνει τα ταξινομημένα δεδομένα της στη διεργασία $P2$. Η $P2$ συγχωνεύει την ταξινομημένη λίστα που λαμβάνει από τη $P1$ με την τοπική της ταξινομημένη λίστα και τελικά στέλνει το πρώτο μισό της συγχωνευμένης λίστας στη διεργασία $P1$. Υπενθυμίζεται ότι ο ακολουθιακός αλγόριθμος για τη συγχώνευση δύο ταξινομημένων λιστών n και m στοιχείων αντίστοιχα απαιτεί $n+m-1$ συγκρίσεις στη χειρότερη περίπτωση.

Εναλλακτικά, σε αντιστοιχία με τον δεύτερο τρόπο υλοποίησης της compare-and-exchange, κάθε διεργασία στέλνει την ταξινομημένη λίστα της στην άλλη διεργασία όπως φαίνεται στο επόμενο σχήμα. Στη συνέχεια και οι δύο διεργασίες εκτελούν τη λειτουργία της συγχώνευσης και κρατούν το τμήμα της συγχωνευμένης λίστας που τους αντιστοιχεί.



9.2 Αλγόριθμος Bubble Sort

Ο αλγόριθμος αυτός εκτελεί $n-1$ περάσματα σε μία λίστα n στοιχείων. Στο πρώτο πέρασμα, το μεγαλύτερο στοιχείο τοποθετείται στο τέλος της λίστας, μετά από μία σειρά συγκρίσεων και εναλλαγών, οι οποίες αρχίζουν από την αρχή της λίστας.

Στο i-στό πέρασμα, οι $i-1$ μεγαλύτεροι αριθμοί έχουν τοποθετηθεί στις τελευταίες θέσεις του πίνακα και ο αλγόριθμος προχωρεί στην ταξινόμηση των $n-i+1$ στοιχείων στις πρώτες θέσεις της λίστας. Στο τέλος του $(n-1)$-οστού περάσματος, η λίστα έχει προκύψει ταξινομημένη.

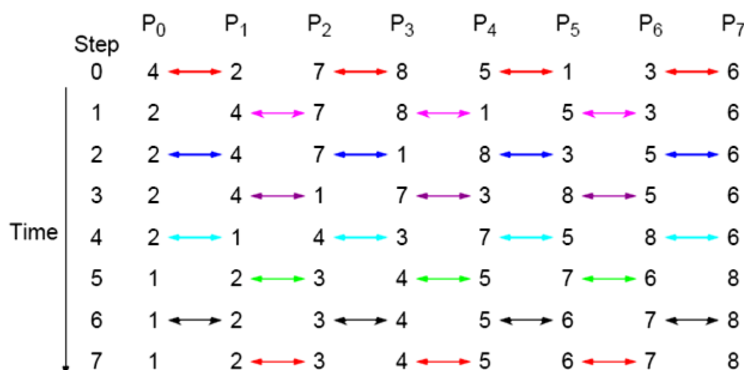
Είναι εύκολο να δει κανείς ότι το πλήθος των λειτουργιών compare-and-exchange είναι $n(n+1)/2$ και επομένως αν κάθε τέτοια λειτουργία απαιτεί χρόνο $O(1)$, η συνολική πολυπλοκότητα θα είναι $O(n^2)$.

9.2.1 Παράλληλος αλγόριθμος Bubble Sort

Χρησιμοποιώντας την τεχνική σωλήνωσης, μπορούν να εκτελεστούν την ίδια χρονική στιγμή λειτουργίες compare-and-exchange από διαφορετικές φάσεις. Η φάση i δεν μπορεί να "προσπεράσει" τη φάση $i-1$ κατά την επεξεργασία της. Επίσης, σε οποιαδήποτε χρονική στιγμή, δεν θα πρέπει το ίδιο στοιχείο να εμπλέκεται σε δύο διαφορετικές λειτουργίες compare-and-exchange.

9.3 Αλγόριθμος ταξινόμησης Odd-Even (Transposition) Sort

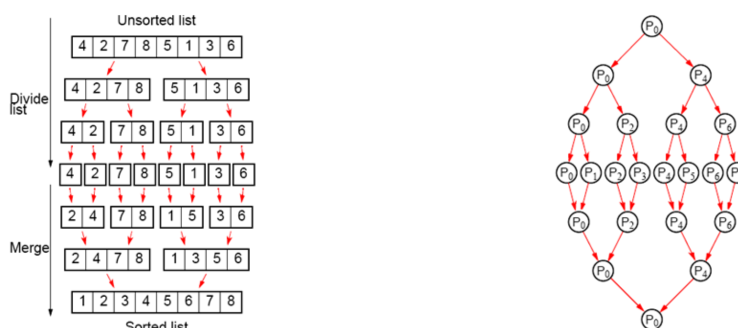
Ο αλγόριθμος αυτός αποτελεί παραλλαγή του bubble sort. Λειτουργεί σε δύο φάσεις που εναλλάσσονται, την ζυγή φάση και την περιττή. Στη ζυγή φάση, οι διεργασίες με ζυγό αναγνωριστικό συγκρίνουν/ανταλλάσουν τα δεδομένα τους με το δεξιό τους γείτονα. Στη μονή φάση, οι διεργασίες με μονό αναγνωριστικό συγκρίνουν/ανταλλάσουν δεδομένα με τον δεξιό τους γείτονα. Ο αλγόριθμος αυτός αποτελείται από n φάσεις συνολικά και κάθε φάση απαιτεί το χρόνο μίας σύγκρισης. Άρα, η πολυπλοκότητα του αλγορίθμου είναι $\Theta(n)$.



9.4 Αλγόριθμος Mergesort

Ο αλγόριθμος mergesort είναι ένας κλασσικός ακολουθιακός αλγόριθμος, ο οποίος χρησιμοποιεί την τακτική διαίρει και βασίλευε. Η μη ταξινομημένη λίστα πρώτα διαιρείται στα δύο. Κάθε μισό επίσης διαιρείται στα δύο και αυτό συνεχίζεται μέχρι να προκύψουν λίστες του ενός στοιχείου. Στη συνέχεια, ζεύγη στοιχείων συγχωνεύονται σε μία ταξινομημένη λίστα δύο αριθμών. Ζεύγη αυτών των λιστών συγχωνεύονται σε λίστες των τεσσάρων αριθμών και στη συνέχεια αυτές οι λίστες συγχωνεύονται σε λίστες των 8 στοιχείων κοκ., μέχρι να προκύψει ολόκληρη η λίστα ταξινομημένη.

Για την παράλληλη υλοποίηση αυτού του αλγορίθμου, οι διεργασίες επικοινωνούν σε δενδρική διάταξη. Αρχικά η διεργασία P_0 έχει όλα τα δεδομένα και δίνει τα μισά μεγαλύτερα στοιχεία στη διεργασία P_4. Στη συνέχεια, οι δύο αυτές διεργασίες στέλνουν το δεύτερο μισό της λίστας που διαθέτουν στις διεργασίες P_2 και P_6 αντίστοιχα κοκ. Οι λειτουργίες της συγχώνευσης ακολουθούν την αντίστροφη πορεία.



Ένα σημαντικό μειονέκτημα αυτής της παράλληλης υλοποίησης είναι ότι ο φόρτος επεξεργασίας δεν κατανέμεται ομοιόμορφα στις διάφορες διεργασίες. Αρχικά, μόνο μία διεργασία είναι ενεργή, στη συνέχεια 2, μετά 4 κοκ. Κατά τη φάση της συγχώνευσης, αυτό το μοτίβο επεξεργασίας αντιστρέφεται.

Η πολυπλοκότητα του ακολουθιακού αλγορίθμου είναι $O(n \log n)$. Ο παράλληλος αλγόριθμος αποτελείται από $2 \log n$ βήματα. Συγκεκριμένα, στη φάση της διαίρεσης, έχουμε τις εξής πολυπλοκότητες επικοινωνίας

$$\begin{array}{ll} t_{\text{startup}} + (n/2)t_{\text{data}} & P_0 \rightarrow P_4 \\ t_{\text{startup}} + (n/4)t_{\text{data}} & P_0 \rightarrow P_2, P_4 \rightarrow P_6 \\ t_{\text{startup}} + (n/8)t_{\text{data}} & P_0 \rightarrow P_1, P_2 \rightarrow P_3, P_4 \rightarrow P_5, P_6 \rightarrow P_7 \\ & \vdots \end{array}$$

Στη φάση της συγχώνευσης θα έχουμε τις εξής πολυπλοκότητες επικοινωνίας:

$$\begin{array}{ll} & \vdots \\ t_{\text{startup}} + (n/8)t_{\text{data}} & P_0 \leftarrow P_1, P_2 \leftarrow P_3, P_4 \leftarrow P_5, P_6 \leftarrow P_7 \\ t_{\text{startup}} + (n/4)t_{\text{data}} & P_0 \leftarrow P_2, P_4 \leftarrow P_6 \\ t_{\text{startup}} + (n/2)t_{\text{data}} & P_0 \leftarrow P_4 \end{array}$$

Αν αθροίσουμε τις παραπάνω πολυπλοκότητες, η συνολική πολυπλοκότητα επικοινωνίας του αλγορίθμου θα είναι

$$t_{\text{comm}} \simeq 2 \log p t_{\text{startup}} + 2nt_{\text{data}}$$

Για την πολυπλοκότητα υπολογισμού, παρατηρούμε ότι συγκρίσεις γίνονται μόνο στη φάση της συγχώνευσης. Στο βήμα i της φάσης συγχώνευσης, κάθε διεργασία εκτελεί συγχώνευση δύο λιστών $2i - 1$ στοιχείων και εκτελεί $2i - 1$ συγκρίσεις για την συγχώνευση αυτή. Αφού η φάση της συγχώνευσης έχει $\log p$ φάσεις συνολικά, η συνολική πολυπλοκότητα θα είναι:

$$t_{\text{comm}} = \sum_{i=1}^{\log p} (2^i - 1) = O(p).$$

Αν το πλήθος των διεργασιών είναι μικρότερο από το πλήθος των στοιχείων για ταξινόμηση, κάθε διεργασία αναλαμβάνει περισσότερα από ένα στοιχεία.

Στον προηγούμενο κώδικα, η συγχώνευση των δύο λιστών εκτελείται από μία μόνο διεργασία. Αυτή η επεξεργασία μπορεί να υλοποιηθεί και αυτή παράλληλα. Έστω $A[1, \dots, m]$, $B[1, \dots, n]$ οι δύο ταξινομημένες λίστες που πρέπει να συγχωνευθούν, έστω ότι $m \geq n$ και $C[1, \dots, m + n]$ το αποτέλεσμα της συγχώνευσης. Αναζητούμε στη λίστα B με δυαδική αναζήτηση το μεσαίο στοιχείο της A, δηλαδή το στοιχείο $A[q]$ με $q = \lfloor (1 + m)/2) \rfloor$. Έστω ότι το στοιχείο $A[q]$ είναι μεταξύ των στοιχείων $B[r]$ και $B[r + 1]$.

Στη συνέχεια το πρόβλημα ανάγεται στην εκτέλεση δύο συγχωνεύσεων ζευγών λιστών. Το πρώτο ζεύγος είναι οι λίστες $A[1, \dots, q - 1]$ και $B[1, \dots, r]$, ενώ το δεύτερο ζεύγος είναι οι λίστες $A[q + 1, \dots, m]$ και $B[r + 1, \dots, n]$. Ο ψευδοκώδικας θα έχει ως εξής:

```
Parallel_merge (A[1...m],B[1...n],C[1...n+m])
    q=int((1+m)/2)
    r=Binary_Search(A[q], B[1..n])
    C[q+r]=A[q]
    Parallel do:
      Parallel_merge(A[1...q-1],B[1...r], C[1,...q+r-1])
      Parallel_merge(A[q+1,...,m],B[r+1,...n], C[q+r+1...m+n])
    End do
```

Ο χρόνος εκτέλεσης της δυαδικής αναζήτησης θα είναι $\Theta(\log n)$. Ο χρόνος $T(n, m)$ για τη συγχώνευση των A και B θα είναι

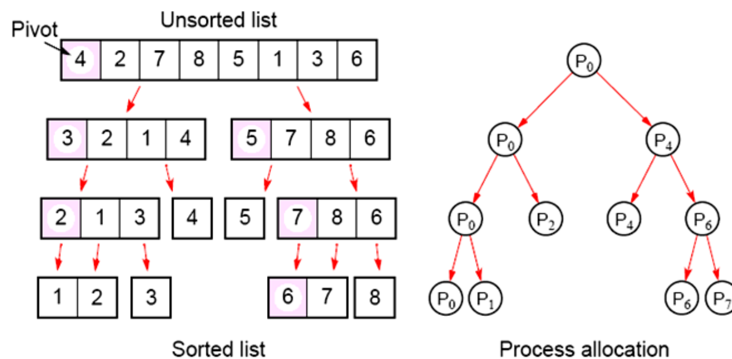
$$T(n, m) = \max\{T(q - 1, r), T(m - q, n - r)\} + \Theta(\log n).$$

Αν $n = m$ μπορεί να αποδειχθεί ότι $T(n, n) = O(\log^2 n)$.

9.5 Ο αλγόριθμος Quicksort

Ο αλγόριθμος αυτός είναι ένας πολύ γνωστός ακολουθιακός αλγόριθμος ταξινόμησης, ο οποίος έχει μέση πολυπλοκότητα χρόνου $O(n \log n)$. Στον αλγόριθμο αυτό, η αρχική λίστα διαιρείται σε δύο υπολίστες. Τα στοιχεία της μίας υπολίστας θα πρέπει να είναι μικρότερα από τα στοιχεία της άλλης υπολίστας. Αυτό επιτυγχάνεται με την επιλογή ενός αριθμού, του στοιχείου οδηγού. Κάθε ένα από τα υπόλοιπα στοιχεία της λίστας συγκρίνεται με το στοιχείο οδηγό και ανάλογα με το αποτέλεσμα της σύγκρισης, τοποθετείται στη μία ή την άλλη υπολίστα. Το στοιχείο οδηγός μπορεί να είναι οποιοδήποτε στοιχείο στη λίστα αλλά συνήθως επιλέγεται το πρώτο στοιχείο. Στη συνέχεια, το στοιχείο οδηγός καταχωρείται σε μία από τις δύο λίστες ή φυλάσσεται και τοποθετείται στη τελική του θέση αφού ολοκληρωθεί η επεξεργασία. Η παραπάνω διαδικασία επαναλαμβάνεται αναδρομικά στις δύο υπολίστες μέχρι να προκύψουν λίστες του ενός στοιχείου. Σε αυτό το σημείο, η διαδοχική τοποθέτηση των λιστών αυτών δίνει και το τελικό ταξινομημένο αποτέλεσμα.

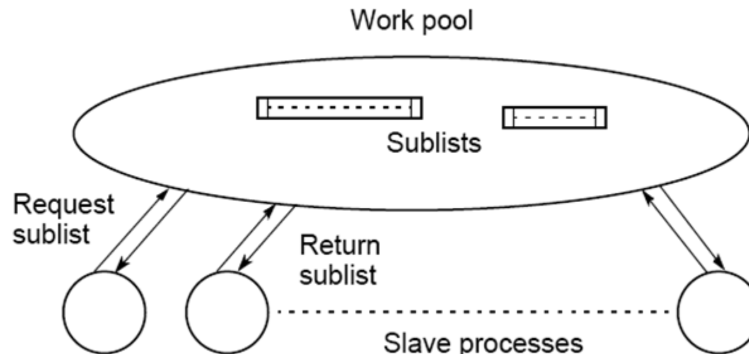
Για την παράλληλη υλοποίηση του αλγορίθμου Quicksort, ανάλογη μέθοδος με αυτή της παράλληλης υλοποίησης του Mergesort ακολουθείται. Συγκεκριμένα, η διεργασία P_0 χωρίζει την αρχική λίστα σε δύο υπολίστες σύμφωνα με το στοιχείο οδηγό. Στη συνέχεια, η P_0 κρατάει την πρώτη υπολίστα που περιέχει τα μικρότερα στοιχεία, και δίνει την δεύτερη υπολίστα στη διεργασία P_4. Στη συνέχεια, οι διεργασίες P_0 και P_4 υποδιαιρούν περαιτέρω τις λίστες τους και δίνουν τη δεύτερη υπολίστα στις διεργασίες P_2 και P_6. Η διαδικασία συνεχίζεται μέχρι να προκύψουν λίστες ενός στοιχείου.



Αν υποθέσουμε ότι σε κάθε βήμα έχουμε ισο-διάσπαση, μπορούμε να δούμε ότι χρόνος υπολογισμού είναι $tcomp \simeq 2n$ και ότι ο χρόνος επικοινωνίας είναι $t_{\text{comm}} \simeq \log p t_{\text{startup}} + n t_{\text{data}}$. Όμως, η ισοδιάσπαση είναι ιδανική υπόθεση και συνήθως οι λίστες που προκύπτουν μετά τη διάσπαση δεν έχουν το ίδιο μέγεθος. Στη χειρότερη περίπτωση αυτό μπορεί να οδηγήσει σε συνολική πολυπλοκότητα $O(n^2)$. Βασικό πρόβλημα τόσο στη παράλληλη υλοποίηση του Mergesort όσο και του Quicksort είναι ότι η αρχική διαίρεση της λίστας σε δύο υπολίστες γίνεται από μία μόνο διεργασία (διεργασία P_0). Αυτό έχει ως αποτέλεσμα τη σημαντική επιβράδυνση ολόκληρου του υπολογισμού.

Η τεχνική της δεξαμενής έργων μπορεί να χρησιμοποιηθεί στην παράλληλη υλοποίηση του Quicksort. Σε αυτή την περίπτωση, οι λίστες που προκύπτουν από τις διασπάσεις είναι τα

διαθέσιμα έργα προς εκτέλεση. Μετά τη διάσπαση, κάθε διεργασία κρατάει τη μία υπολίστα και επιστρέφει την άλλη στη δεξαμενή έργων. Όταν, μία διεργασία ολοκληρώσει την επεξεργασία στη λίστα της, παίρνει την επόμενη διαθέσιμη υπολίστα από τη δεξαμενή έργων.

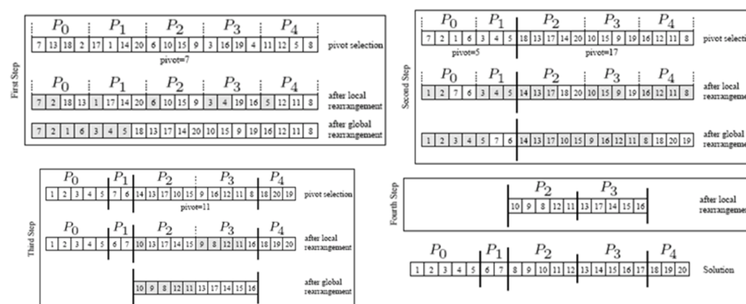


9.5.1 Εναλλακτικός τρόπος υλοποίησης του Quicksort σε σύστημα διαμοιραζόμενης μνήμης

Θεωρούμε μία λίστα μεγέθους n, η οποία κατανέμεται ομοιόμορφα σε p διεργασίες. Μία από τις διεργασίες επιλέγει ένα στοιχείο οδηγό και ενημερώνει τις υπόλοιπες διεργασίες για την τιμή του.

- (Local rearrangement.) Κάθε διεργασία χωρίζει τη λίστα σε δύο, L_i και U_i, με βάση την τιμή του στοιχείου οδηγού.
- (Global rearrangement.) Τα στοιχεία στη διαμοιραζόμενη μνήμη αναδιατάσσονται κατά τέτοιο τρόπο ώστε όλες οι λίστες L_i να τοποθετηθούν σε συνεχόμενες θέσεις μνήμης και ανάλογα πράττουμε για τις λιστες U_i.

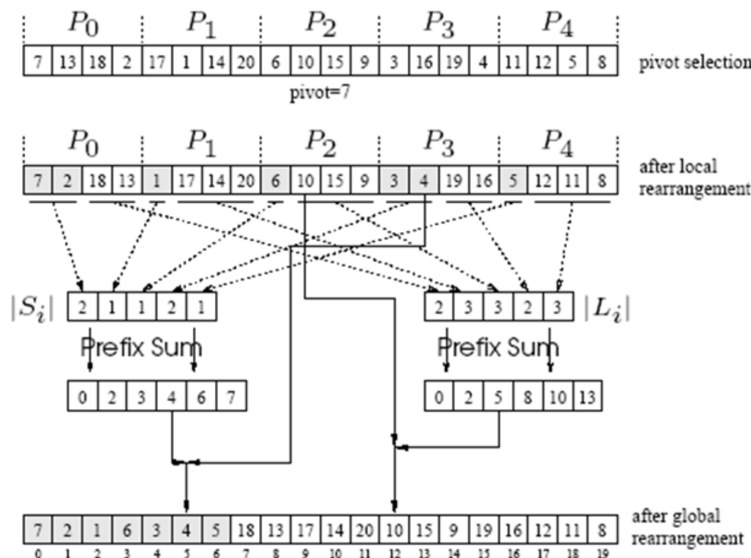
Έστω L η λίστα που δημιουργείται από αυτή την τοποθέτηση των λιστών L_i και U η αντίστοιχη λίστα για τις λίστες U_i. Το σύνολο των επεξεργαστών χωρίζονται σε δύο ομάδες (σε αναλογία με το μέγεθος των λιστών L και U). Στη συνέχεια, η παραπάνω διαδικασία εφαρμόζεται αναδρομικά στις λίστες L και U. Η αναδρομή ολοκληρώνεται όταν οι λίστες L και U καταχωρηθούν σε μία μόνο διεργασία. Σε αυτή την περίπτωση, η διεργασία ταξινομεί τη λίστα L ή την U χρησιμοποιώντας τον ακολουθιακό Quicksort.



Σχήμα 9.1: Παράδειγμα εκτέλεσης.

Δεν έχουμε περιγράψει την συνένωση των λιστών L_i και U_i στις λίστες L και U αντίστοιχα. Το πρόβλημα είναι ο προσδιορισμός της σωστής θέσης κάθε στοιχείου στην τελική συνενωμένη λίστα. Κάθε διεργασία υπολογίζει το πλήθος των στοιχείων της τοπικής λίστας που είναι

μικρότερα και αντίστοιχα μεγαλύτερα από το στοιχείο οδηγό. Στη συνέχεια, εκτελείται ένας υπολογισμός prefix-sum και προσδιορίζεται με αυτό τον τρόπο η θέση κάθε στοιχείου στη λίστες L και U. Από τη στιγμή που αυτές οι θέσεις είναι γνωστές, κάθε διεργασία μπορεί να τοποθετήσει τα δεδομένα της στις λίστες L και U.



Σχήμα 9.2: Παράδειγμα εκτέλεσης.

Αν υποθέσουμε ότι σε κάθε βήμα οι λίστες ισοδιασπώνται γύρω από το στοιχείο οδηγό, μπορούμε να αποδείξουμε ότι συνολική πολυπλοκότητα του αλγορίθμου θα είναι:

$$T(n,p) = \Theta\left(\frac{n}{p}\log\frac{n}{p}\right) = \Theta(\log^2 p).$$

9.6 Ο Αλγόριθμος Odd-Even Mergesort

Ο αλγόριθμος Odd-Even Mergesort βασίζεται στη ρουτίνα odd-even merge η οποία συγχωνεύει δύο λίστες n στοιχείων η κάθε μία. Συγκεκριμένα, αν δύο λίστες εισόδου είναι οι $a_1, a_2, a_3, \ldots, a_n$ και $b_1, b_2, b_3, \ldots, b_n$, όπου n είναι δύναμη του 2, ο odd-even merge αλγόριθμος εκτελεί τις ακόλουθες ενέργειες:

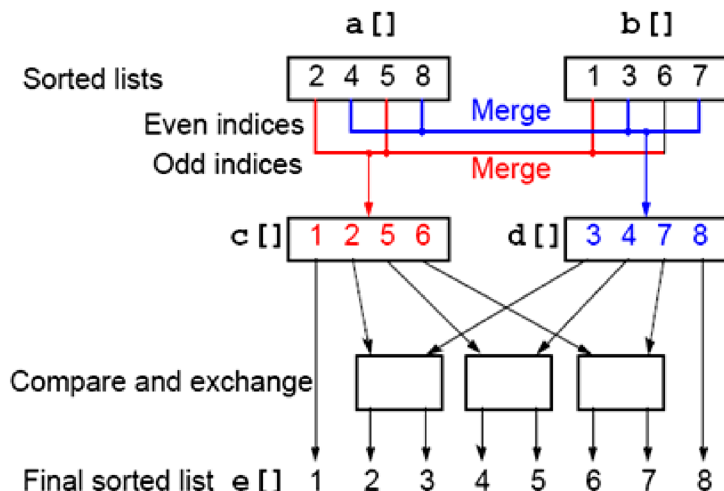
1. Τα στοιχεία με μονούς δείκτες από κάθε λίστα, δηλαδή τα $a_1, a_3, a_5, \ldots, a_{n-1}$ και $b_1, b_3, b_5 \ldots, b_{n-1}$ συγχωνεύονται σε μία ταξινομημένη λίστα $c_1, c_2, c_3, \ldots, c_n$.

2. Τα στοιχεία με ζυγούς δείκτες από κάθε λίστα, δηλαδή τα $a_2, a_4, a_6, \ldots, a_n$ και $b_2, b_4, b_6, \ldots, b_n$ συγχωνεύονται σε μία ταξινομημένη λίστα $d_1, d_2, d_3, \ldots, d_n$.

3. Η τελική ταξινομημένη λίστα $e_1, e_2, \ldots, e_{2n}$ προκύπτει ως εξής:

$$e_{2i} = \min\{c_{i+1}, d_i\}, \qquad e_{2i+1} = \max\{c_{i+1}, d_i\}.$$

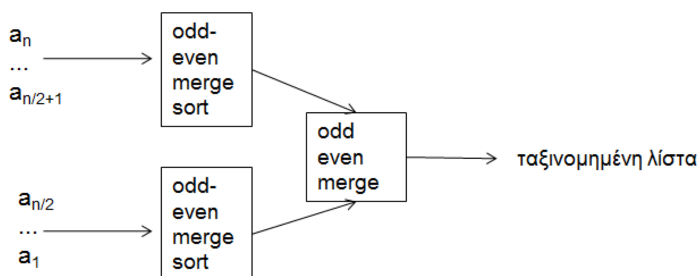
Η συγχώνευση στα βήματα 1 και 2 πραγματοποιείται με αναδρομική εκτέλεση του odd-even merge αλγόριθμου.



Με n επεξεργαστές, η συνολική πολυπλοκότητα του odd-even merge αλγόριθμου θα είναι $O(\log n)$.

ο αλγόριθμος αυτός μπορεί να ορισθεί αναδρομικά ως εξής:

```
odd-even-mergesort (a_1, a_2, ..., a_n) begin
    if n > 1 then
        return odd-even-merge (odd-even-mergesort (a_1, a_2, ..., a_{n/2}),
        odd-even-mergesort (a_{n/2+1}, ..., a_n)) ;
    end if
    else
        return a_n;
    end if
end
```



Η πολυπλοκότητα του Odd-Even Mergesort δίνεται από την ακόλουθη αναδρομική σχέση:

$$T(n) = T(n/2) + \Theta(\log n),$$

που έχει συνολική πολυπλοκότητα $\Theta(\log^2 n)$.

9.7 Ο Αλγόριθμος Bitonic Mergesort

Ο αλγόριθμος αυτός βασίζεται στις ιδιότητες των διτονικών (bitonic) ακολουθιών. Καταρχήν, διτονική χαρακτηρίζεται μία ακολουθία αριθμών $a_0, a_1, \ldots, a_{n-1}$, όταν αποτελείται από δύο υπακολουθίες $a_0, a_1, \ldots, a_i$ και $a_i, a_{i+1}, a_{i+2}, \ldots, a_{n-1}$, όπου η πρώτη υπακολουθία είναι αύξουσα και

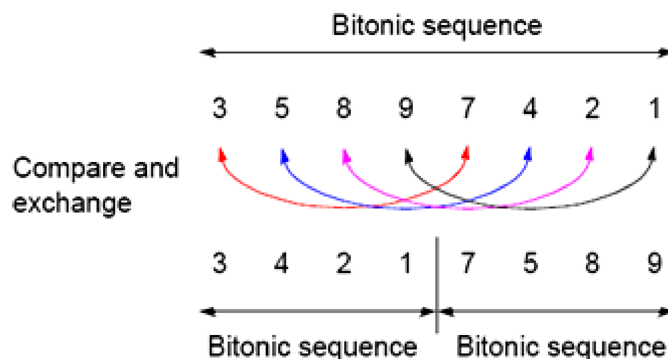
η δεύτερη φθίνουσα, δηλαδή

$$a_0 < a_1 < a_2 < a_3 < \cdots < a_{i-1} < a_i > a_{i+1} > \cdots > a_{n-2} > a_{n-1}$$

για κάποια τιμή i, με $0 \le i < n$.

Μία ακολουθία χαρακτηρίζεται επίσης διτονική όταν μπορεί να προέλθει από τη παραπάνω ακολουθία με μία κυκλική ολίσθηση (αριστερή ή δεξιά). Για παράδειγμα, η ακολουθία $(2, 4, 6, 5, 3, 1)$ είναι διτονική ακολουθία καθώς και η ακολουθία $(3, 1, 2, 4, 6, 5)$, αφού μπορεί να προκύψει από την πρώτη με κυκλική ολισθηση.

Κάθε διτονική ακολουθία $A = (a_0, a_1, \ldots, a_{n-1})$ έχει την εξής σημαντική ιδιότητα: Αν $c_i = \min\{a_i, a_{i+n/2}\}$ και $d_i = \max\{a_i, a_{i+n/2}\}$, όπου $i = 0, \ldots, n/2-1$, τότε οι ακολουθίες $C = (c_0, c_1, \ldots, c_{n/2-1})$ και $D = (d_0, d_1, \ldots, d_{n/2-1})$ είναι διτονικές και όλα τα στοιχεία της C είναι μικρότερα από τα στοιχεία της D.



Σχήμα 9.3: Παράδειγμα.

Αφού η παραπάνω διαδικασία τοποθετεί τα μικρότερα στοιχεία στη C και τα μεγαλύτερα στοιχεία στη D, αν επαναλάβουμε τη διαδικασία αυτή αναδρομικά στις λίστες C και D, τελικά θα λάβουμε τα στοιχεία της A ταξινομημένα.

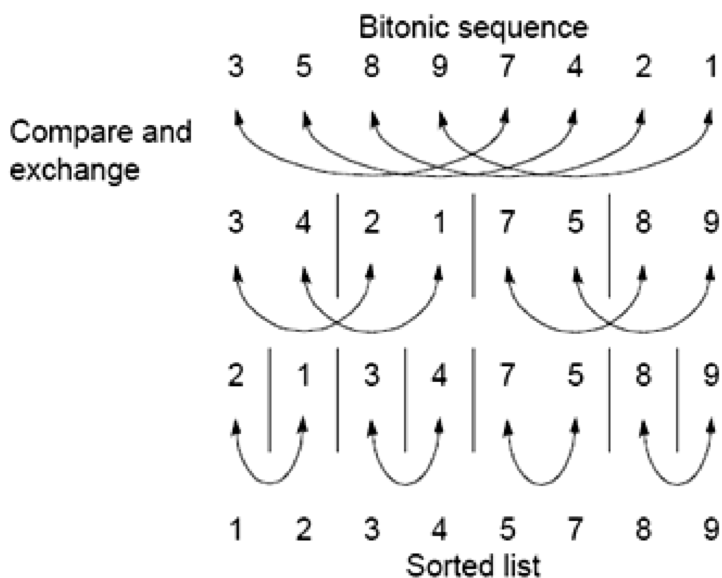
Η ταξινόμηση μίας διτονικής ακολουθίας n στοιχείων με n διεργασίες/επεξεργαστές απαιτεί χρόνο $O(\log n)$. Ο αλγόριθμος ταξινόμησης μίας διτονικής ακολουθίας μπορεί να χρησιμοποιηθεί για την ταξινόμηση οποιαδήποτε ακολουθίας αριθμών. Η βασική ιδέα είναι να συγχωνεύουμε τα στοιχεία σε όλο και μεγαλύτερες διτονικές ακολουθίες, αρχίζοντας από τα γειτονικά στοιχεία της ακολουθίας. Αρχικά, τα γειτονικά στοιχεία σχηματίζουν αύξουσες και φθίνουσες ακολουθίες στοιχείων. Έτσι, με τη χρήση του αλγόριθμου ταξινόμησης διτονικών ακολουθιών, ζεύγη γειτονικών ακολουθιών μπορούν να δώσουν εναλλασόμενες αύξουσες και φθίνουσες ακολουθίες τεσσάρων στοιχείων. Επαναλαμβάνοντας την παραπάνω διαδικασία, προκύπτουν διτονικές ακολουθίες αυξανόμενου μήκους. Στο τελικό βήμα, η μοναδική διτονική ακολουθία ταξινομείται με τον αλγόριθμο ταξινόμησης διτονικών ακολουθιών.

Συνολικά έχουμε k εφαρμογές του αλγόριθμου ταξινόμησης διτονικών ακολουθιών ($n = 2k$). Στην i-οστή εφαρμογή του, ο αλγόριθμος εκτελείται σε i βήματα. Το συνολικό πλήθος βημάτων θα είναι

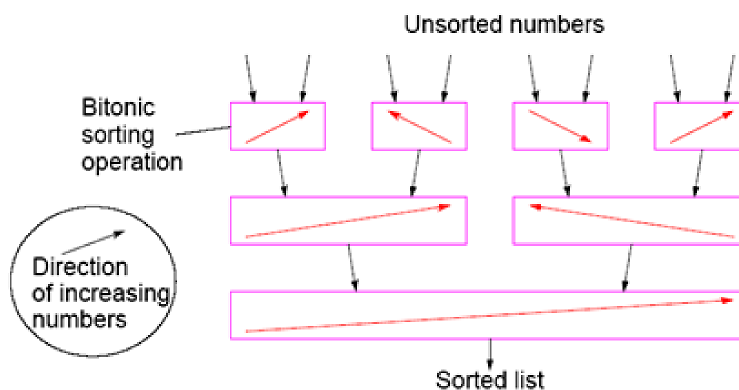
$$1 + 2 + \cdots + k = k(k + 1)/2 = \Theta(\log^2 n).$$

9.8 Αλγόριθμοι Bucket sort και Sample sort

Βασική υπόθεση στον αλγόριθμο Bucket sort είναι ότι τα στοιχεία του πίνακα εισόδου παίρνουν τιμές σε ένα συγκεκριμένο διάστημα τιμών $[a, b]$, $l = b - a + 1$. Το διάστημα αυτό υπο-



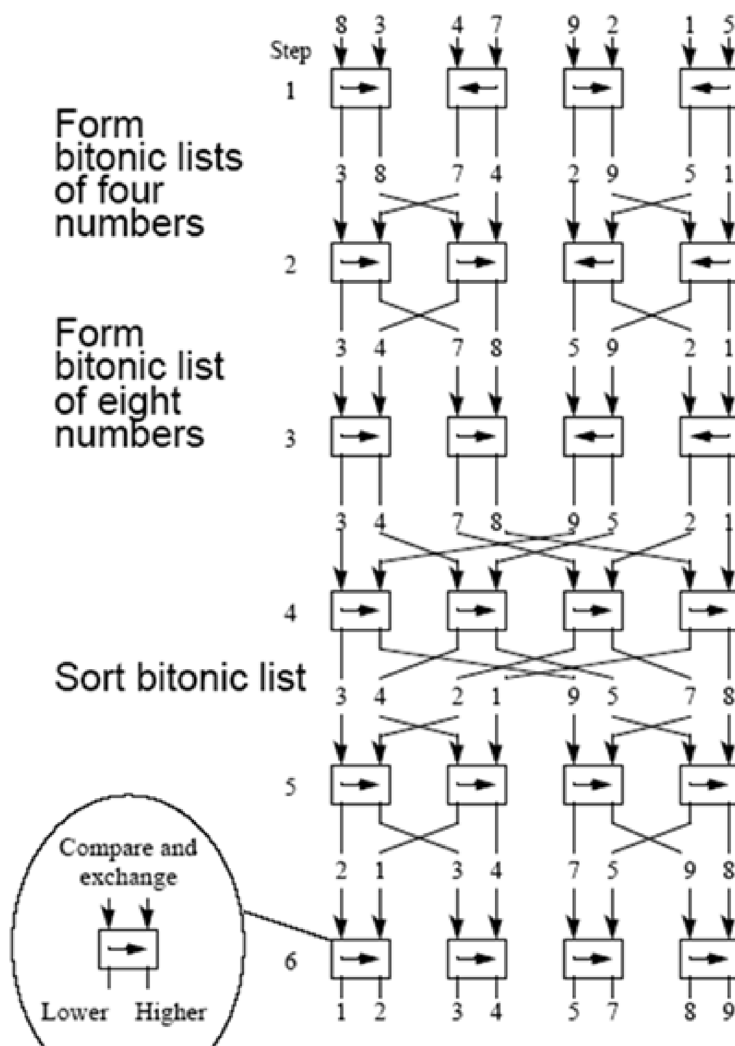
Σχήμα 9.4: Παράδειγμα ταξινόμησης διτονικής ακολουθίας 8 στοιχείων.



Σχήμα 9.5: Γενικός αλγόριθμος ταξινόμησης με βάση τον αλγόριθμο ταξινόμησης διτονικής ακολουθίας.

διαιρείται σε m διαστήματα (buckets) ίσου μήκους. Έτσι, ο i-οστός κάδος περιέχει στοιχεία στο διάστημα $[(i-1)l/m, il/m-1]$, $i = 1, \ldots, m$. Κάθε στοιχείο ανάλογα με την τιμή του τοποθετείται στο κατάλληλο κάδο (bucket). Αν τα στοιχεία κατανέμονται ομοιόμορφα στο διάστημα $[a, b]$, οι κάδοι αναμένονται να έχουν περίπου το ίδιο πλήθος στοιχείων. Τα στοιχεία κάθε κάδου ταξινομούνται τοπικά και στη συνέχεια το ταξινομημένο περιεχόμενο κάθε κάδου τοποθετούνται στο τελικό πίνακα. Πρώτα τοποθετούνται τα περιεχόμενα του 1ου κάδου, στη συνέχεια του 2ου κοκ. Ο συνολικός χρόνος του ακολουθιακού αλγορίθμου θα είναι $\Theta(n\log(n/m))$, εφόσον ισχύει η υπόθεση της ισοκατανομής των στοιχείων στους m κάδους.

Η παράλληλη υλοποίηση του bucket sort είναι σχετικά απλή. Αν έχουμε p διεργασίες, επιλέγουμε $m = p$, δηλαδή ο κάδος i αντιστοιχεί στη διεργασία/επεξεργαστή i. Επομένως, κάθε διεργασία είναι υπεύθυνη για ένα συγκεκριμένο εύρος τιμών. Κάθε διεργασία χωρίζει τα στοιχεία της σύμφωνα με τους m κάδους. Τα στοιχεία του i-οστού κάδου στέλνονται στην διαδικασία i. Επομένως, το βήμα αυτό απαιτεί επικοινωνία κάθε διεργασίας με όλες τις υπόλοιπες και αποστολή διαφορετικών δεδομένων σε κάθε διεργασία. Μετά το βήμα επικοινωνίας, κάθε διερ-



Σχήμα 9.6: Παράδειγμα ταξινόμισης 8 στοιχείων.

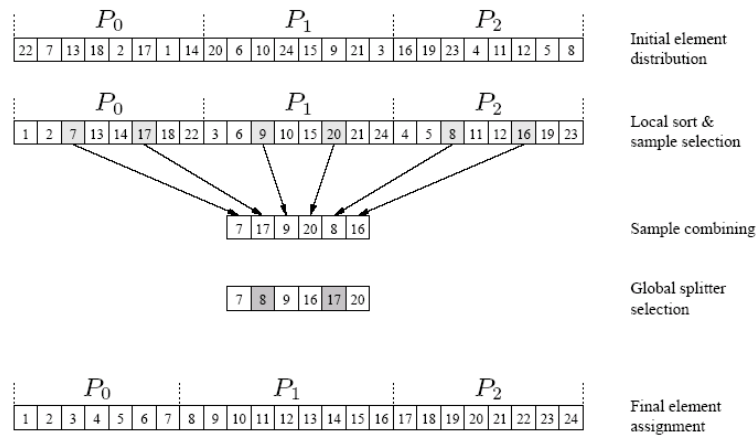
γασία i θα έχει στη διάθεση της όλα τα στοιχεία του αρχικού πίνακα που ανήκουν στο κάδο i. Στη συνέχεια, κάθε διεργασία ταξινομεί τα στοιχεία αυτά και ο τελικός ταξινομημένος πίνακας προκύπτει κατανεμημένος στις διεργασίες.

Η διαίρεση του εύρους τιμών $[a, b]$ σε m ίσου μήκους διαστήματα, δεν οδηγεί συνήθως σε ισοκατανομή των στοιχείων του πίνακα στους διάφορους κάδους. Ο αλγόριθμος sample sort αντιμετωπίζει αυτό το πρόβλημα, ορίζοντας διαφορετικά το εύρος τιμών κάθε κάδου. Συγκεκριμένα, θα πρέπει να επιλεγούν κατά τέτοιο τρόπο τα όρια των κάδων (στοιχεία-splitters) έτσι ώστε σε κάθε κάδο να καταλήξουν το ίδιο περίπου πλήθος στοιχείων με μεγάλη πιθανότητα. Η επιλογή των splitters γίνεται με τον εξής τρόπο:

- Αρχικά ο πίνακας των n στοιχείων διαιρείται σε p βλοςκς των $n/p = l$ στοιχείων το καθένα, όπου p είναι το πλήθος διαθέσιμων διεργασιών. Η διεργασία i αναλαμβάνει το i-οστό block, $i = 1, \dots, p$.

- Κάθε διεργασία ταξινομεί το δικό της block $[b_1, b_2, \dots, b_l]$, με εφαρμογή του αλγόριθμου quicksort και στη συνέχεια επιλέγει $p - 1$ στοιχεία, συγκεκριμένα τα στοιχεία $i(l/p)$, $i = 1, \dots, p - 1$.

- Συνολικά, υπάρχουν $p(p-1)$ στοιχεία που έχουν επιλεχθεί με αυτή τη διαδικασία. Όλα αυτά τα στοιχεία συλλέγονται σε μία διεργασία η οποία και τα ταξινομεί.
- Στη συνέχεια, η παραπάνω διεργασία επιλέγει πάλι ομοιόμορφα $p-1$ στοιχεία από τα $p(p-1)$ ταξινομημένα στοιχεία και τα στέλνει στις υπόλοιπες διεργασίες.
- Τα στοιχεία αυτά αποτελούν τα στοιχεία-splitters που ορίζουν τους p κάδους. Μπορεί να αποδειχθεί ότι κάθε ένα από τα p buckets θα έχει το πολύ $2n/p$ στοιχεία.
- Ο υπόλοιπος αλγόριθμος ακολουθεί τη λογική του bucket sort, δηλαδή συγκέντρωση από κάθε διεργασία των στοιχείων που ανήκουν στον κάδο που έχει αναλάβει και ταξινόμηση των στοιχείων των κάδων.



Σχήμα 9.7: Παράδειγμα.

Στο παραπάνω παράδειγμα, εφαρμόζεται ο sample sort για την ταξινόμηση ενός πίνακα 24 στοιχείων. Σε πρώτη φάση, κάθε διεργασία αφού ταξινομήσει τα στοιχεία της, επιλέγει δύο splitters. Οι έξι συνολικά splitters συλλέγονται σε μία διεργασία, η οποία τα ταξινομεί και στη συνέχεια επιλέγει τους δύο τελικούς splitters (στοιχεία 8 και 17). Κάθε διεργασία συλλέγει από τις υπόλοιπες τα στοιχεία του κάδου που της αντιστοιχεί και στη συνέχεια τα ταξινομεί.

Η πολυπλοκότητα του παράλληλου sample sort μπορεί να προσδιορισθεί ως εξής: Η εσωτερική ταξινόμηση των n/p στοιχείων απαιτεί χρόνο $\Theta((n/p)\log(n/p))$, και η επιλογή των $p-1$ στοιχείων απαιτεί χρόνο $\Theta(p)$. Ο χρόνος για την συλλογή των $p(p-1)$ στοιχείων θέλει χρόνο $\Theta(p^2)$, ο χρόνος για την τοπική ταξινόμηση των στοιχείων αυτών είναι $\Theta(p^2 \log p)$, και η επιλογή πάλι p"1 στοιχείων απαιτεί χρόνο $\Theta(p)$.

Κάθε διεργασία μπορεί να χωρίσει τα στοιχεία της σε p κάδους αναζητώντας τους $p-1$ splitters στον τοπικό του ταξινομημένο πίνακα με $p-1$ δυαδικές αναζητήσεις και σε συνολικό χρόνο $\Theta(p\log(n/p))$. Ο χρόνος για την αποστολή των στοιχείων κάθε κάδου από κάθε διεργασία στη διεργασία που έχει αναλάβει τον συγκεκριμένο κάδο απαιτεί συνολικά χρόνο $O(n/p)$.

Συνολικά, η πολυπλοκότητα του αλγορίθμου θα είναι:

$$T = \Theta(n/p \log n/p + p^2 \log p + p \log n/p + n/p).$$

Κεφάλαιο 10

Αριθμητικοί Αλγόριθμοι

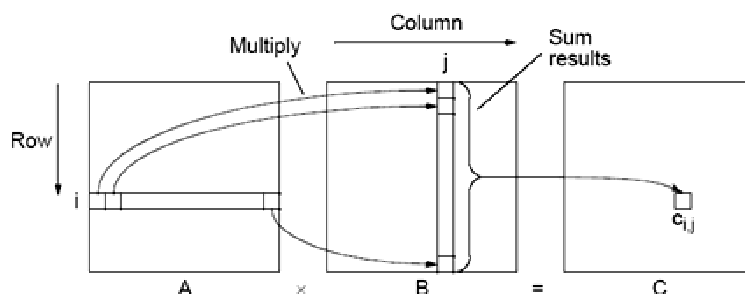
10.1 Άθροιση και Πολλαπλασιασμός Πινάκων

Το άθροισμα $C = A + B$ δύο $n \times n$ πινάκων A και B είναι ένας $n \times n$ πίνακας, προκύπτει από το άθροισμα των αντίστοιχων στοιχείων των A και B, δηλαδή

$$c_{i,j} = a_{i,j} + b_{i,j}, \qquad 0 \le i < n, 0 \le j < n.$$

Το γινόμενο $C = AB$ δύο πινάκων A και B διαστάσεων $n \times l$ και $l \times m$ είναι ένας $n \times m$ πίνακας και ορίζεται ως εξής:

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}, \qquad 0 \le i < n, 0 \le j < m.$$

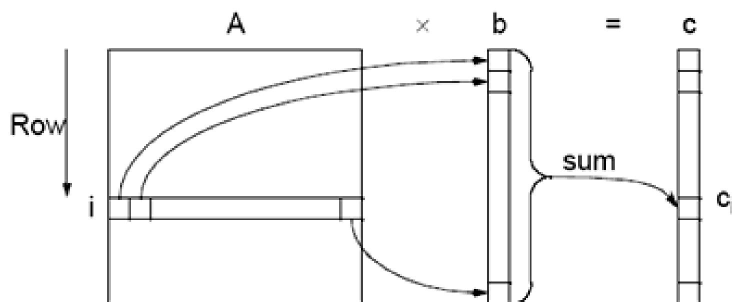


Σχήμα 10.1: Πολλαπλασιασμός πινάκων.

Ο πολλαπλασιασμός πίνακα με διάνυσμα προκύπτει από το προηγούμενο ορισμό του γινόμενου πινάκων, αν θεωρήσουμε το B μονοδιάστατο πίνακα $m = 1$.

Είναι επίσης γνωστό ότι ένα σύστημα γραμμικών εξισώσεων μπορεί να γραφεί υπό μορφή πινάκων ως $Ax = b$, όπου x είναι το διάνυσμα των αγνώστων, A είναι ο πίνακας των συντελεστών των αγνώστων και b είναι το διάνυσμα των σταθερών των εξισώσεων. Ο ακολουθιακός κώδικας για τον πολλαπλασιασμό των πινάκων έχει ως εξής:

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



Σχήμα 10.2: Πολλαπλασιασμός πίνακα με διάνυσμα.

Ο αλγόριθμος αυτός απαιτεί n^3 πολλαπλασιασμούς και n^3 προσθέσεις και επομένως η πολυπλοκότητα του αλγορίθμου θα είναι $O(n^3)$. Ο παραπάνω υπολογισμός μπορεί εύκολα να παραλληλοποιηθεί, αφού δεν υπάρχουν εξαρτήσεις μεταξύ των επαναλήψεων των δύο εξωτερικών βρόχων. Έτσι:

- Με n επεξεργαστές, κάθε επεξεργαστής μπορεί να αναλάβει μία επανάληψη του εξωτερικού βρόχου. Κάθε επεξεργαστής εκτελεί $O(n^2)$ πράξεις.
- Με n^2 επεξεργαστές, κάθε επεξεργαστής αναλαμβάνει τον υπολογισμό ενός στοιχείου του πίνακα C. Κάθε επεξεργαστής εκτελεί $O(n)$ πράξεις.
- Με n^3 επεξεργαστές, μπορούμε να υλοποιήσουμε παράλληλα και τον υπολογισμό κάθε στοιχείου του πίνακα C. Σε αυτή την περίπτωση, η πολυπλοκότητα του παράλληλου αλγόριθμου είναι $O(\log n)$.

Μπορεί να αποδειχθεί ότι το κάτω όριο πολυπλοκότητας για τον παράλληλο πολλαπλασιασμό πινάκων είναι $\Omega(\log n)$. Πρέπει να σημειωθεί ότι οι παραπάνω πολυπλοκότητες δεν περιλαμβάνουν το χρόνο επικοινωνίας που απαιτείται, αλλά αντιστοιχούν μόνο σε χρόνο επεξεργασίας.

10.2 Χωρισμός Πινάκων σε Υποπίνακες

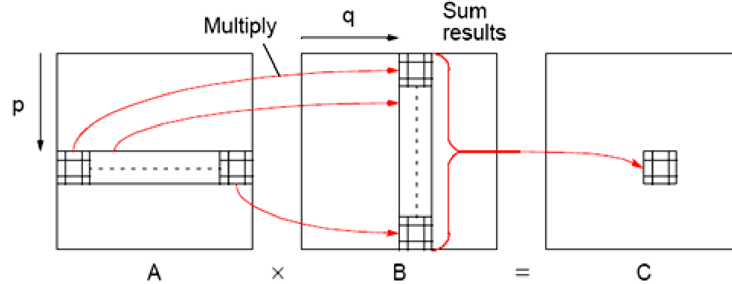
Στην πράξη, έχουμε στη διάθεση μας πολύ λιγότερους από n επεξεργαστές. Σε αυτή τη περίπτωση κάθε επεξεργαστής αναλαμβάνει τον υπολογισμό περισσότερων στοιχείων του πίνακα C. Συγκεκριμένα, αν έχουμε s^2 επεξεργαστές, μπορούμε να χωρίζουμε κάθε πίνακα σε s^2 υποπίνακες διαστάσεων $n/s \times n/s$ ελεμεντς. Αν $A_{p,q}$ είναι ο υποπίνακας στη γραμμή p και στη στήλη q του block πίνακα A, το γινόμενο $C = AB$ μπορεί να υπολογισθεί ως εξής:

```
for (p = 0; p < s; p++)
   for (q = 0; q < s; q++) {
      C[p][q] = 0;                              /* clear elements of submatrix */
      for (r = 0; r < m; r++)                   /* submatrix multiplication &*/
          C[p][q] = C[p][q] + A[p][r] * B[r][q];   /*add to accum. submatrix*/
}
```

Αξίζει να σημειωθεί ότι στον υπολογισμό

```
C[p][q] = C[p][q] + A[p][r] * B[r][q];
```

όλες οι πράξεις είναι μεταξύ πινάκων διαστάσεων $n/s \times n/s$. Αυτός ο τρόπος πολλαπλασιασμού πινάκων είναι γνωστός ως πολλαπλασιασμός block πινάκων.



Για παράδειγμα για τον υπολογισμό του γινομένου δύο πινάκων A και B διαστάσεων 4×4, μπορούμε να χωρίσουμε τους πίνακες σε υποπίνακες 2×2 και στη συνέχεια να εκτελέσουμε τον υπολογισμό σε block πίνακες:

$$\left[\begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right] \times \left[\begin{array}{cc|cc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array}\right]$$

Έτσι, ο αρχικός υπολογισμός ανάγεται ουσιαστικά στην πρόσθεση και τον πολλαπλασιασμό πινάκων 2×2. Για παράδειγμα, ο υπολογισμός του στοιχείου $C_{0,0}$ σύμφωνα με τον τύπο $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0}$ συνεπάγεται στους ακόλουθους υπολογισμούς:

$$\overset{A_{0,0}}{\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}} \times \overset{B_{0,0}}{\begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix}} + \overset{A_{0,1}}{\begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix}} \times \overset{B_{1,0}}{\begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

Στην παράλληλη υλοποίηση του πολλαπλασιασμού πινάκων, θεωρούμε ότι υπάρχουν s^2 διεργασίες/επεξεργαστές $P_{i,j}$, $i, j = 0, \ldots, s-1$, και κάθε διεργασία αναλαμβάνει τον υπολογισμό του υποπίνακα $C_{i,j}$ του πίνακα C. Για τον υπολογισμό αυτόν, χρειάζεται τους πίνακες $A_{i,k}$ και $B_{k,j}$, $k = 0, \ldots, s-1$, αφού ο πίνακας $C_{i,j}$ δίνεται από τη σχέση

$$C_{i,j} = A_{i,0}B_{0,j} + A_{i,1}B_{1,j} + \cdots + A_{i,s-1}B_{s-1,j}.$$

Ο υπολογισμός αυτός απαιτεί s πολλαπλασιασμούς πινάκων διαστάσεων $n/s \times n/s$. Κάθε τέτοιος πολλαπλασιασμός πινάκων εκτελεί $(n/s)^3$ απλούς πολλαπλασιασμούς και επομένως ο υπολογισμός του υποπίνακα $C_{i,j}$ απαιτεί $s(n/s)^3 = n^3/s^2$ απλούς πολλαπλασιασμούς.

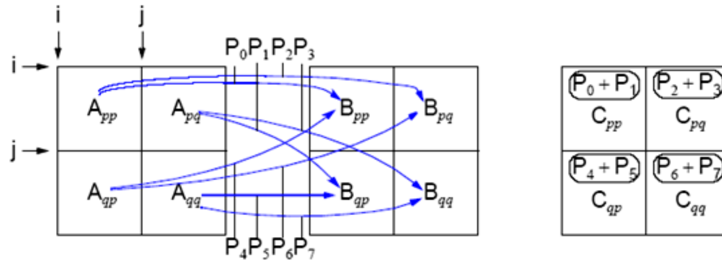
Όσον αφορά το χρόνο επικοινωνίας, θεωρούμε ότι και οι δύο πίνακες A και B είναι αρχικά αποθηκευμένοι στη master διεργασία. Στη συνέχεια τα στοιχεία αυτά θα πρέπει να μοιραστούν στις s^2 slave διεργασίες. Με δύο διαφορετικά μηνύματα από τη master διεργασία, κάθε slave διεργασία θα λάβει $s(n/s)^2$ στοιχεία από τον πίνακα A και $s(n/s)^2$ στοιχεία από το πίνακα B, συνολικά $2n^2/s$ στοιχεία.

Μετά τον υπολογισμό των υποπινάκων $C_{i,j}$, τα $(n/s)^2$ στοιχεία κάθε υποπίνακα θα πρέπει να σταλούν πίσω στη master διεργασία. Έτσι, συνολικά ο χρόνος επικοινωνίας θα είναι

$$t_{\text{comm}} = 2s^2 t_{\text{startup}} + 2n^2 s t_{\text{data}} + s^2 t_{\text{startup}} + n^2 t_{\text{data}} = 3s^2 t_{\text{startup}} + (2s+1)n^2 t_{\text{data}}.$$

10.2.1 Αναδρομική υλοποίηση του πολλαπλασιασμού πινάκων

Όπως αναφέρθηκε προηγουμένως, ο πολλαπλασιασμός δύο πινάκων $n \times n$ μπορεί να αναχθεί σε πολλάπλασιασμό και πρόσθεση πινάκων $n/s \times n/s$, αν διαιρέσουμε κάθε πίνακα σε $s \times s$ υποπίνακες. Αυτή η ιδέα μπορεί να εφαρμοσθεί αναδρομικά με μία τεχνική διαίρει και βασίλευε. Αρχικά, οι δύο $n \times n$ πίνακες A και B χωρίζονται σε 4 υποπίνακες. θεωρούμε ότι η διάσταση n είναι δύναμη του δύο.



Ο υπολογισμός ανάγεται στον υπολογισμό των 8 γινομένων $A_{i,k}B_{k,j}$, όπου $i, j, k = p, q$. Οι πίνακες σε αυτά τα γινόμενα είναι διαστάσεων $n/4 \times n/4$ και μπορούν να υπολογιστούν με την ίδια λογική, χωρίζοντας τους πίνακες αυτούς σε υποπίνακες $n/8 \times n/8$. Ακολουθώντας αυτή την αναδρομική λογική, θα φτάσουμε σε πίνακες 1×1, όπου ο πολλαπλασιασμός μπορεί να υλοποιηθεί με προφανή τρόπο.

Ο αναδρομικός αλγόριθμος για τον πολλαπλασιασμό πινάκων μπορεί να διατυπωθεί ως εξής:

```
mat_mult(App, Bpp, s)
{
if (s == 1)           /* if submatrix has one element */
    C = A * B;        /* multiply elements */
else {                /* continue to make recursive calls */
    s = s/2;          /* no of elements in each row/column */
    P0 = mat_mult(App, Bpp, s);
    P1 = mat_mult(Apq, Bqp, s);
    P2 = mat_mult(App, Bpq, s);
    P3 = mat_mult(Apq, Bqq, s);
    P4 = mat_mult(Aqp, Bpp, s);
    P5 = mat_mult(Aqq, Bqp, s);
    P6 = mat_mult(Aqp, Bpq, s);
    P7 = mat_mult(Aqq, Bqq, s);
    Cpp = P0 + P1;      /* add submatrix products to */
    Cpq = P2 + P3;      /* form submatrices of final matrix */
    Cqp = P4 + P5;
    Cqq = P6 + P7;
}
return (C);             /* return final matrix */
}
```

Αυτός ο αναδρομικός υπολογισμός του γινομένου δύο πινάκων μπορεί εύκολα να υλοποιηθεί παράλληλα. Συγκεκριμένα, κάθε διεργασία/επεξεργαστής μπορεί να αναλάβει τον υπολογισμό μίας από τις οκτώ αναδρομικές κλήσεις της συνάρτησης. Είναι επίσης δυνατόν κάθε αναδρομική κλήση να εκτελεσθεί από περισσότερες από μία διεργασίες. Γενικά, μπορούμε να συνεχίσουμε την εκτέλεση της αναδρομής, μέχρι το πλήθος των αναδρομικών κλήσεων να φθάσει το πλήθος των διαθέσιμων επεξεργαστών/διεργασιών. Ιδανικά, το πλήθος των διαθέσιμων διεργασιών θα πρέπει να είναι δύναμη του 8.

Ο αλγόριθμος αυτός είναι κατάλληλος για υλοποίηση σε συστήματα κατανεμημένης μνήμης, αφού καθώς προχωράει η αναδρομή, οι προσβάσεις στα δεδομένα του πίνακα A και B έχουν τοπικό χαρακτήρα και επομένως οι περισσότερες από αυτές τις προσβάσεις ικανοποιούνται από τις τοπικές μνήμες cache των επεξεργαστών.

10.2.2 Ο αλγόριθμος του Cannon για πολλαπλασιασμό πινάκων

Ο αλγόριθμος υποθέτει ότι το διασυνδετικό δίκτυο μεταξύ των επεξεργαστών είναι κυκλικό πλέγμα (torus). Η διάταξη του torus μπορεί εύκολα να προσομοιωθεί και σε άλλα διασυνδετικά δίκτυα. Για παράδειγμα, στο MPI μπορούμε εύκολα να προγραμματίσουμε την τοπολογία mesh. Στην πράξη όμως, η υλοποίηση του πλέγματος σε μία αρχιτεκτονική cluster, όπου όλοι οι υπολογιστές επικοινωνούν μέσω ενός κοινού διαύλου (π.χ. Ethernet), οδηγεί σε συμφόρηση στο κοινό μέσο λόγω των ταυτόχρονων μεταδόσεων που υπαγορεύει η αρχιτεκτονική του δικτύου mesh. Σε αυτήν την περίπτωση, το μέγεθος των υποπινάκων που χρησιμοποιεί ο αλγόριθμος Cannon πρέπει να είναι σχετικά μεγάλο, έτσι ώστε ο όγκος επεξεργασίας που εκτελεί κάθε επεξεργαστής μεταξύ διαδοχικών επικοινωνιών να αντισταθμίζει την επιβάρυνση λόγω επικοινωνιών.

Θα περιγράψουμε τον αλγόριθμο θεωρώντας ότι έχουμε στη διάθεση μας n^2 επεξεργαστές, όσα και τα στοιχεία των πινάκων A, B και C. Όταν έχουμε λιγότερους επεξεργαστές, η διαδικασία που ακολουθεί μπορεί εύκολα να προσαρμοσθεί, μόνο που αντί για απλά στοιχεία, η επεξεργασία γίνεται σε υποπίνακες των πινάκων A, B και C.

1. Αρχικά ο επεξεργαστής $P_{i,j}$ έχει τα στοιχεία $a_{i,j}$ και $b_{i,j}$, $0 \le i < n$, $0 \le k < n$.

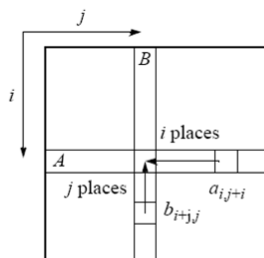
2. Τα στοιχεία των πινάκων A και B μετακινούνται κατά τέτοιο τρόπο, έτσι ώστε στοιχεία των A και B που μπορούν να πολλαπλασιαστούν μεταξύ τους να βρεθούν στον ίδιο επεξεργαστή. Συγκεκριμένα, η i-οστή γραμμή του A ολισθαίνει κατά i θέσεις αριστερά και η j-οστή στήλη του B ολισθαίνει j θέσεις προς τα πάνω. Αυτό έχει ως αποτέλεσμα τα στοιχεία $a_{i,j+i}$ και $b_{i+j,j}$ να βρεθούν στον επεξεργαστή $P_{i,j}$. Αυτά τα στοιχεία είναι χρήσιμα για τον υπολογισμό του στοιχείου $c_{i,j}$ του πίνακα C.

3. Κάθε επεξεργαστής $P_{i,j}$ πολλαπλασιάζει τα στοιχεία που έχει τοπικά.

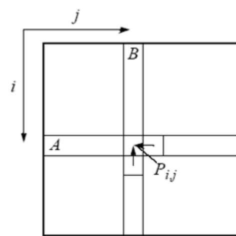
4. Η i-οστή γραμμή του A ολισθαίνει μία θέση προς τα αριστερά και η j-οστή στήλη του B ολισθαίνει μία θέση προς τα πάνω. Αυτό έχει ως αποτέλεσμα κάθε επεξεργαστής να έχει δυο νέα στοιχεία των A και B που μπορούν να πολλαπλασιασθούν μεταξύ τους.

5. Κάθε επεξεργαστής $P_{i,j}$ πολλαπλασιάζει αυτά τα στοιχεία και το αποτέλεσμα προστίθεται στην τρέχουσα τιμή του στοιχείου $c_{i,j}$.

6. Τα βήματα 4 και 5 επαναλαμβάνονται, μέχρι το τελικό αποτέλεσμα να προκύψει ($n - 1$ οριζόντιες και κατακόρυφες ολισθήσεις απαιτούνται, όταν οι πίνακες είναι $n \times n$).



Βήμα 2 – Ευθυγράμμιση των στοιχείων Α και Β



Βήμα 4 – Ολίσθηση των στοιχείων των Α και Β κατά μία θέση

Θα αναλύσουμε την πολυπλοκότητα του αλγορίθμου όταν έχουμε $s \times s$ επεξεργαστές:

- Επικοινωνία: Η αρχική ευθυγράμμιση απαιτεί $s-1$ ολισθήσεις των στοιχείων των A και B. Κατά την εκτέλεση του αλγορίθμου, θα έχουμε άλλες $s-1$ ολισθήσεις των στοιχείων των A και B. Κάθε ολίσθηση μεταφέρει $(n/s)^2$ στοιχεία. Άρα συνολικά θα έχουμε:

$$t_{\text{comm}} = 4(s-1)(t_{\text{startup}} + (n/s)^2 t_{\text{data}})$$

- Υπολογισμός: Κάθε πολλαπλασιασμός και άθροιση υποπινάκων απαιτεί $(n/s)^3$ πολλαπλασιασμούς και αθροίσεις. Συνολικά θα έχουμε

$$t_{\text{comp}} = 2s(n/s)^3 = 2n^3/s^2$$

10.3 Επίλυση συστήματος γραμμικών εξισώσεων

Έστω ένα σύστημα γραμμικών εξισώσεων:

$$\begin{array}{llll}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 & \cdots & + a_{n-1,n-1}x_{n-1} & = b_{n-1} \\
 & \vdots & & \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 & \cdots & + a_{2,n-1}x_{n-1} & = b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 & \cdots & + a_{1,n-1}x_{n-1} & = b_1 \\
a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 & \cdots & + a_{0,n-1}x_{n-1} & = b_0
\end{array}$$

το οποίο σε μορφή πινάκων γράφεται ως $Ax = b$.

Ο στόχος είναι η εύρεση τιμών για τους αγνώστους $x_0, x_1, \ldots, x_{n-1}$, με δεδομένες τιμές των $a_{0,0}, a_{0,1}, \ldots, a_{n-1,n-1}$ και $b_0, \ldots, b_n$. Πρώτα θα δούμε τρόπους επίλυσης του γραμμικού συστήματος όταν ο πίνακας A είναι πυκνός, δηλαδή όταν τα περισσότερα στοιχεία του είναι μη μηδενικά. Στη συνέχεια, θα δούμε μεθόδους επίλυσης του γραμμικού συστήματος όταν ο πίνακας A είναι αραιός.

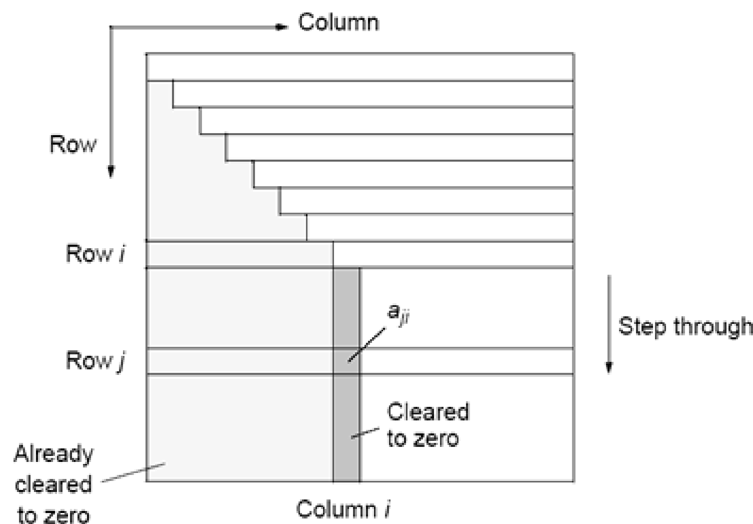
10.3.1 Επίλυση γραμμικού συστήματος με τη μέθοδο Gaussian Elimination

Ο στόχος αυτής της μεθόδου είναι η μετατροπή ενός γενικού συστήματος γραμμικών εξισώσεων σε ένα τριγωνικό σύστημα εξισώσεων. Στη συνέχεια, το σύστημα αυτό επιλύεται με την προς τα πίσω αντικατάσταση των αγνώστων (Back Substitution). Βασίζεται στο γεγονός ότι η τελική λύση ενός γραμμικού συστήματος δεν επηρεάζεται όταν σε οποιαδήποτε γραμμή προσθέσουμε κάποια άλλη γραμμή πολλαπλασιασμένη με μία σταθερά. Η διαδικασία αρχίζει

από τη πρώτη γραμμή και επισκέπτεται κάθε γραμμή μέχρι και την τελευταία. Κατά την επίσκεψη της i-οστής γραμμής, αντικαθιστούμε κάθε γραμμή j κάτω από την i-οστή γραμμή με τη (γραμμή j) + (γραμμή i)$a_{j,i}/a_{i,i}$. Επομένως, μετά τις παραπάνω αντικαταστάσεις, όλα τα στοιχεία του πίνακα A που είναι επί της j-οστής στήλης και κάτω από την i-οστή γραμμή μηδενίζονται, αφού

$$a_{j,i} = a_{j,i} + a_{i,i}\left(\frac{-a_{j,i}}{a_{i,i}}\right) = 0.$$

Στο επόμενο σχήμα, φαίνεται ο μηδενισμός αυτών των στοιχείων κατά την επίσκεψη στην i-οστή γραμμή του πίνακα A. Επίσης, όλα τα στοιχεία που είναι σε στήλες πιο αριστερά της στήλης j και κάτω από τη διαγώνιο, έχουν ήδη μηδενιστεί από επισκέψεις σε προηγούμενες γραμμές του πίνακα.



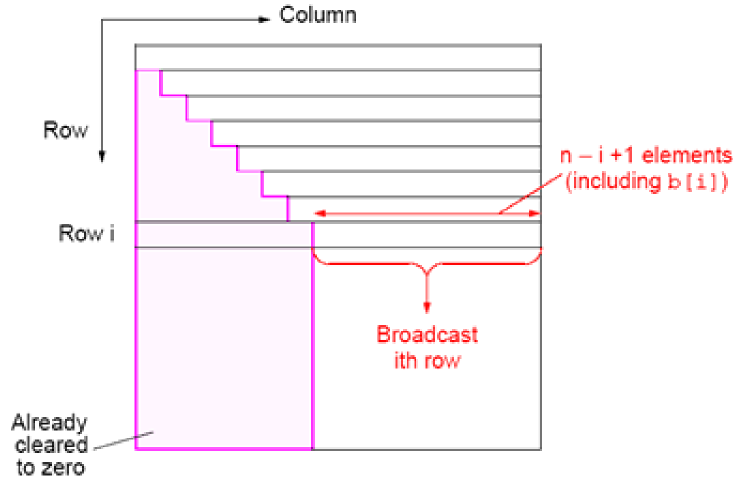
Αν κατά την επίσκεψη στην i-οστή γραμμή, το στοιχείο $a_{i,i}$ είναι μηδέν ή κοντά στο μηδέν, δεν θα μπορέσουμε να υπολογίσουμε την ποσότητα $-a_{j,i}/a_{i,i}$. Η διαδικασία θα πρέπει να τροποποιηθεί εφαρμόζοντας την τεχνική partial pivoting. Συγκεκριμένα, το πρόβλημα επιλύεται με την αντιμετάθεση της i-οστής γραμμής με εκείνη τη γραμμή που είναι κάτω από την i και η οποία έχει το μεγαλύτερο σε απόλυτη τιμή στοιχείο στη στήλη j. Αξίζει να σημειωθεί ότι η επαναδιάταξη εξισώσεων δεν επηρεάζει τη λύση του γραμμικού συστήματος.

Ο ακολουθιακός κώδικας για την τεχνική Gaussian elimination χωρίς partial pivoting είναι ο εξής:

```
for (i = 0; i < n-1; i++)                      /* for each row, except last */
          for (j = i+1; j < n; j++) {          /*step thro subsequent rows */
               m = a[j][i]/a[i][i];                  /* Compute multiplier */
               for (k = i; k < n; k++)         /*last n-i-1 elements of row j*/
                  a[j][k] = a[j][k] - a[i][k] * m;
               b[j] = b[j] - b[i] * m;                /* modify right side */
}
```

Η συνολική πολυπλοκότητα του παραπάνω αλγορίθμου είναι $O(n^3)$.

Ο εξωτερικός βρόχος στο προηγούμενο πρόγραμμα είναι δύσκολο να παραλληλοποιηθεί, γιατί η επεξεργασία σε κάθε επανάληψη εξαρτάται από τα αποτελέσματα όλων των προηγούμενων. Αντίθετα, οι επαναλήψεις του εσωτερικού βρόχου μπορούν να παραλληλοποιηθούν, αφού κάθε



Σχήμα 10.3: Παράλληλη υλοποίηση.

επανάληψη περιλαμβάνει υπολογισμούς σε διαφορετικές γραμμές του πίνακα. Έτσι, αν έχουμε n επεξεργαστές, κάθε επεξεργαστής μπορεί να αναλάβει από μία γραμμή του πίνακα. Συγκεκριμένα, ο επεξεργαστής P_i αναλαμβάνει την i-οστή εξίσωση. Κατά την i-οστή επανάληψη του εξωτερικού βρόχου, θα πρέπει η διεργασία P_i (που έχει αναλάβει την i-οστή γραμμή) να στείλει τα περιεχόμενα της i-οστής εξίσωσης σε όλους τους επεξεργαστές P_j, όπου $j > i$. Στη συνέχεια, κάθε επεξεργαστής $j > i$, υπολογίζει τοπικά τις νέες τιμές των στοιχείων της i-οστής εξίσωσης. Είναι επίσης φανερό ότι όλοι οι επεξεργαστές P_j, με $j < i$, είναι αδρανείς κατά την i-οστή επανάληψη.

Στη συνέχεια, θα προσδιορίσουμε την πολυπλοκότητα της παράλληλης υλοποίησης.

- Επικοινωνία: Συνολικά εκτελούνται $n-1$ λειτουργίες εκπομπής. Στην i-οστή λειτουργία, ο επεξεργαστής P_i στέλνει $n-i+1$ στοιχεία της i-οστής γραμμής στις υπόλοιπες διεργασίες, $i = 1, \ldots, n$). Συνολικά θα έχουμε:

$$t_{\text{comm}} = \sum_{i=1}^{n-1} \left(t_{\text{startup}} + (n-i+1)t_{\text{data}}\right) = (n-1)t_{\text{startup}} + \left(\frac{(n+2)(n+1)}{2} - 3\right)t_{\text{data}} = O(n^2).$$

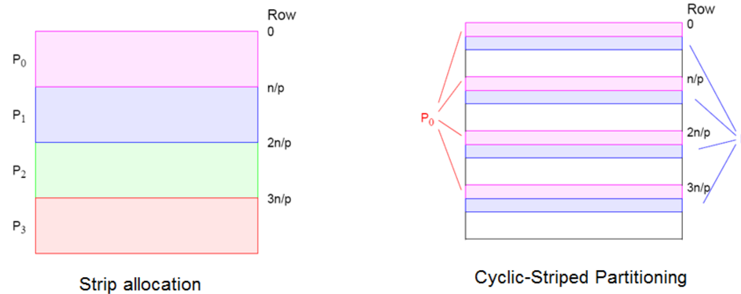
- Υπολογισμός: Μετά από την λήψη των στοιχείων της γραμμής i που έστειλε ο επεξεργαστής P_i, κάθε επεξεργαστής P_j $(j > i)$ εκτελεί $n-j+2$ πολλαπλασιασμούς και $n-j+2$ αφαιρέσεις για τον υπολογισμό των νέων τιμών της j-οστής γραμμής. Συνολικά θα έχουμε

$$t_{\text{comp}} = 2\sum_{j=1}^{n-1}(n-j+2) = \frac{(n+2)(n+1)}{2} - 3 = O(n^2).$$

Το βασικό πρόβλημα με αυτή την υλοποίηση είναι ότι προοδευτικά όλο και περισσότεροι επεξεργαστές είναι αδρανείς. Συγκεκριμένα, αν η τρέχουσα επανάληψη είναι στη i-οστή γραμμή, όλοι οι επεξεργαστές P_j $(j < i)$ είναι αδρανείς.

Στην πράξη, το πλήθος των διαθέσιμων επεξεργαστών είναι πολύ λιγότερο από το πλήθος n των εξισώσεων. Σε αυτή την περίπτωση, κάθε επεξεργαστής αναλαμβάνει περισσότερες από μία εξισώσεις (γραμμές) του γραμμικού συστήματος. Υπάρχουν δύο εναλλακτικές:

Κατανομή κατά λωρίδες (Strip allocation): Ο επεξεργαστής P_i αναλαμβάνει τις γραμμές $i(n/p), \ldots, (i+1)(n/p) - 1$. Η τεχνική αυτή έχει πάλι το μειονέκτημα του προοδευτικά αυξανόμενου αριθμού επεξεργαστών που είναι αδρανείς.



Κατανομή κατά λωρίδες (Strip allocation): Ο επεξεργαστής P_i αναλαμβάνει τις γραμμές $i(n/p), \ldots, (i+1)(n/p) - 1$. Η τεχνική αυτή έχει πάλι το μειονέκτημα του προοδευτικά αυξανόμενου αριθμού επεξεργαστών που είναι αδρανείς.

10.3.2 Επαναληπτικές μέθοδοι επίλυσης γραμμικών συστημάτων

Με χρήση n επεξεργαστών, η χρονική πολυπλοκότητα των απευθείας μεθόδων είναι $O(n^2)$, η οποία είναι σχετικά υψηλή. Η χρονική πολυπλοκότητα μίας επαναληπτικής μεθόδου εξαρτάται από διάφορους παράγοντες όπως:

- τύπος επαναληπτικής μεθόδου,
- πλήθος επαναλήψεων,
- πλήθος αγνώστων και
- απαιτούμενη ακρίβεια.

Οι επαναληπτικές μέθοδοι είναι προτιμότερες όταν ο πίνακας A των συντελεστών του γραμμικού συστήματος είναι αραιός πίνακας. Η πρώτη επαναληπτική μέθοδος είναι η επανάληψη Jacobi. Στην τεχνική αυτή, η i-οστή εξίσωση επιλύεται ως προς τον άγνωστο x_i και η εξίσωση αυτή χρησιμοποιείται για την ενημέρωση της τιμής του αγνώστου x_i. Συγκεκριμένα, η ενημέρωση γίνεται με τον ακόλουθο τύπο:

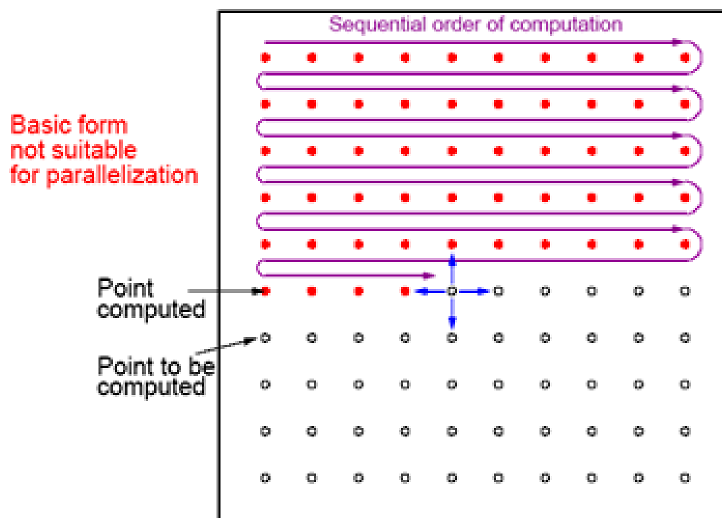
$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1} \right),$$

όπου x_i^k είναι η τιμή του αγνώστου x_i μετά την k-οστή επανάληψη, και x_j^{k-1} είναι η τιμή του αγνώστου x_j μετά την $(k-1)$-οστή επανάληψη. Έχει παρατηρηθεί ότι η σύγκλιση της τεχνικής Jacobi είναι σχετικά αργή και εξαρτάται από τον πίνακα συντελεστών A.

Η δεύτερη επαναληπτική τεχνική είναι η Gauss-Seidel τεχνική, η οποία παρουσιάζει ταχύτερη σύγκλιση. Στην τεχνική αυτή, οι τιμές των αγνώστων που έχουν υπολογισθεί στη τρέχουσα επανάληψη, χρησιμοποιούνται για τον υπολογισμό των επόμενων αγνώστων. Συγκεκριμένα, η ενημέρωση της τιμής του αγνώστου x_i γίνεται με τον ακόλουθο τύπο:

$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^k - \sum_{j=i+1}^{n} a_{i,j} x_j^{k-1} \right).$$

Η επαναληπτική μέθοδος είναι εγγενώς ακολουθιακή, αφού η ενημέρωση της τιμής του αγνώστου x_i εξαρτάται από τις νέες τιμές των αγνώστων x_j ($j < i$). Αντιθέτως, στην τεχνική Jacobi, όλες οι ενημερώσεις των αγνώστων σε μία επανάληψη μπορούν να γίνουν παράλληλα, αφού εξαρτώνται μόνο από τις τιμές των αγνώστων από την προηγούμενη επανάληψη.



10.4 Διάταξη Red-Black

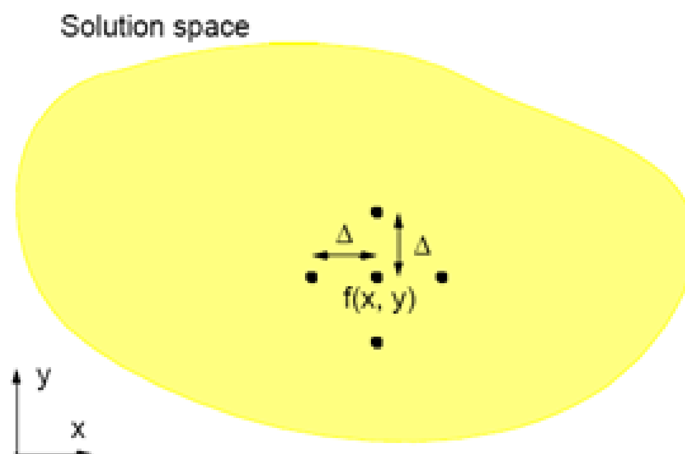
Πολλές φορές, αραιά γραμμικά συστήματα εξισώσεων προκύπτουν ως διακριτή προσέγγιση μερικών διαφορικών εξισώσεων. Για παράδειγμα, με τη «διακριτοποίηση» του επιπέδου, η μερική διαφορική εξίσωση

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

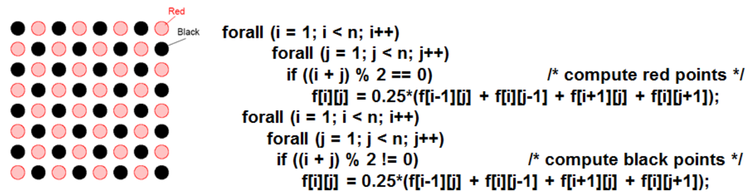
μπορεί να επιλυθεί προσεγγιστικά με τους ακόλουθους επαναληπτικούς τύπους:

$$f_{i,j}^{k} = \frac{1}{4}\left(f_{i-1,j}^{k-1} + f_{i,j-1}^{k-1} + f_{i+1,j}^{k-1} + f_{i,j+1}^{k-1}\right),$$

όπου $f_{i,j}$ είναι η τιμή της άγνωστης συνάρτησης στο σημείο ($i\Delta x$, $j\Delta y$).



Σε τέτοιου είδους γραμμικά συστήματα, τα οποία έχουν προκύψει κατά την επίλυση μερικών διαφορικών εξισώσεων στο επίπεδο, υπάρχει ένας διαφορετικός τρόπος επίσκεψης των αγνώστων, ο οποίος επιτρέπει τη παράλληλη εκτέλεση περισσότερων ενημερώσεων των τιμών των αγνώστων, σε αντίθεση με τη μέθοδο Gauss-Seidel. Συγκεκριμένα, οι άγνωστοι-σημεία στο επίπεδο χωρίζονται σε δύο κατηγορίες: τα κόκκινα και τα μαύρα σημεία. Όλες οι ενημερώσεις στα σημεία του ίδιου χρώματος μπορούν να γίνουν παράλληλα, αφού βασίζονται αποκλειστικά στις τιμές των σημείων του άλλου χρώματος. Έτσι, ο αλγόριθμος εναλλάσσει δύο φάσεις, όπου σε κάθε φάση ενημερώνει τις τιμές στα σημεία του ίδιου χρώματος.



Κεφάλαιο 11

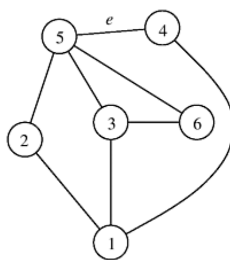
Αλγόριθμοι Γραφημάτων

11.1 Ορισμοί

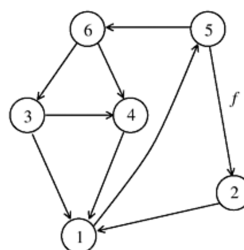
Ένα μη κατευθυνόμενο γράφημα G είναι ένα ζεύγος (V, E), όπου V είναι ένα πεπερασμένο σύνολο σημείων, τα οποία καλούνται κορυφές και E είναι ένα πεπερασμένο σύνολο ακμών. Μία ακμή $e \in E$ είναι ένα μη διατεταγμένο ζεύγος $\{u, v\}$, όπου $u, v \in V$. Σε ένα κατευθυνόμενο γράφημα, οι ακμή e είναι ένα διατεταγμένο ζεύγος (u, v).

Ένα μονοπάτι από μία κορυφή v σε μία κορυφή u είναι μία ακολουθία $(v_0, v_1, v_2, \ldots, v_k)$ κορυφών, όπου $v_0 = v$, $v_k = u$, και $(v_i, v_{i+1}) \in E$ για $i = 0, 1, \ldots, k-1$. Το μήκος ενός μονοπατιού ορίζεται ως το πλήθος των ακμών στο μονοπάτι.

Ένα μη κατευθυνόμενο γράφημα είναι συνεκτικό αν κάθε ζεύγος κορυφών συνδέεται από ένα μονοπάτι. Δάσος είναι ένα άκυκλο γράφημα και δέντρο ένα συνεκτικό άκυκλο γράφημα. Ένα γράφημα όπου κάθε ακμή συσχετίζεται με ένα βάρος λέγεται βεβαρημένο γράφημα.



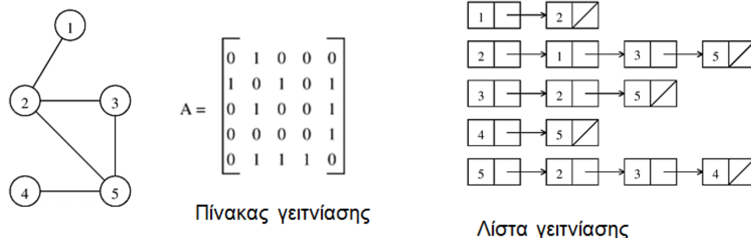
Μη κατευθυνόμενο γράφημα



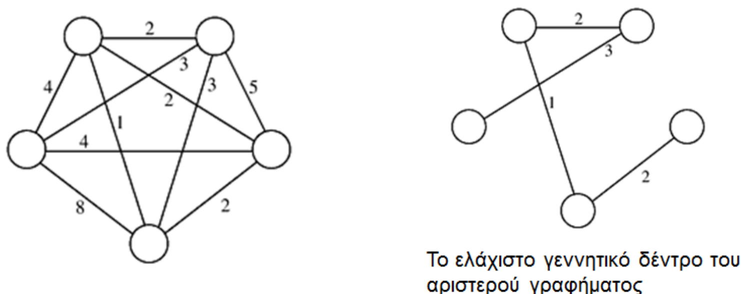
Κατευθυνόμενο γράφημα

Τα γραφήματα μπορούν να αναπαρασταθούν με ένα πίνακα γειτνίασης ή με μία λίστα γειτνίασης. Ο πίνακας γειτνίασης έχει τιμή $a_{i,j} = 1$, αν οι κόμβοι i και j συνδέονται με μία ακμή, διαφορετικά η τιμή του στοιχείου είναι 0. Στην περίπτωση των βεβαρημένων γραφημάτων, $a_{i,j} = w_{i,j}$, δηλαδή το βάρος της ακμής.

Η λίστα γειτνίασης ενός γραφήματος $G = (V, E)$ είναι ένας πίνακας $A[1 \ldots |V|]$ λιστών. Κάθε λίστα $A[v]$ είναι η λίστα όλων των κορυφών που είναι γειτονικές με τη κορυφή v. Για ένα γράφημα με n κορυφές, ο πίνακας γειτνίασης απαιτεί $\Theta(n^2)$ χώρο, ενώ η λίστα γειτνίασης απαιτεί χώρο $\Theta(|E|)$.

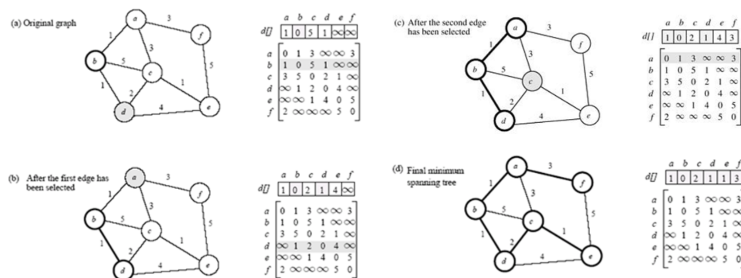


Γεννητικό δέντρο ενός μη κατευθυνόμενου γραφήματος G είναι ένα συνεκτικό άκυκλο υπογράφημα του G, που περιέχει όλες τις κορυφές του G. Πιο απλά, είναι ένα δέντρο το οποίο περιλαμβάνει όλους τις κορυφές του G. Σε ένα βεβαρημένο γράφημα, το βάρος του υπογραφήματος είναι το άθροισμα των βαρών των ακμών του υπογραφήματος. Ελάχιστο Γεννητικό Δέντρο (ΕΓΔ) για ένα βεβαρημένο μη κατευθυνόμενο γράφημα είναι ένα γεννητικό δέντρο με ελάχιστο βάρος.



11.2 Ο αλγόριθμος του Prim

Ο αλγόριθμος του Prim ακολουθεί την τεχνική της απληστίας για την εύρεση του ελάχιστου γεννητικού δέντρου. Αρχικά, ο αλγόριθμος επιλέγει αυθαίρετα μία κορυφή του γραφήματος και την εισάγει στο αρχικά άδειο ΕΓΔ. Σε κάθε βήμα στο υπό κατασκευή ΕΓΔ εισάγεται η κορυφή εκείνη η οποία είναι πιο κοντά στις κορυφές που είναι ήδη στο ΕΓΔ. Στη συνέχεια ακολουθεί ένα παράδειγμα εφαρμογής τού αλγορίθμου Prim.



Σχήμα 11.1: Παράδειγμα - Ο αλγόριθμος του Prim.

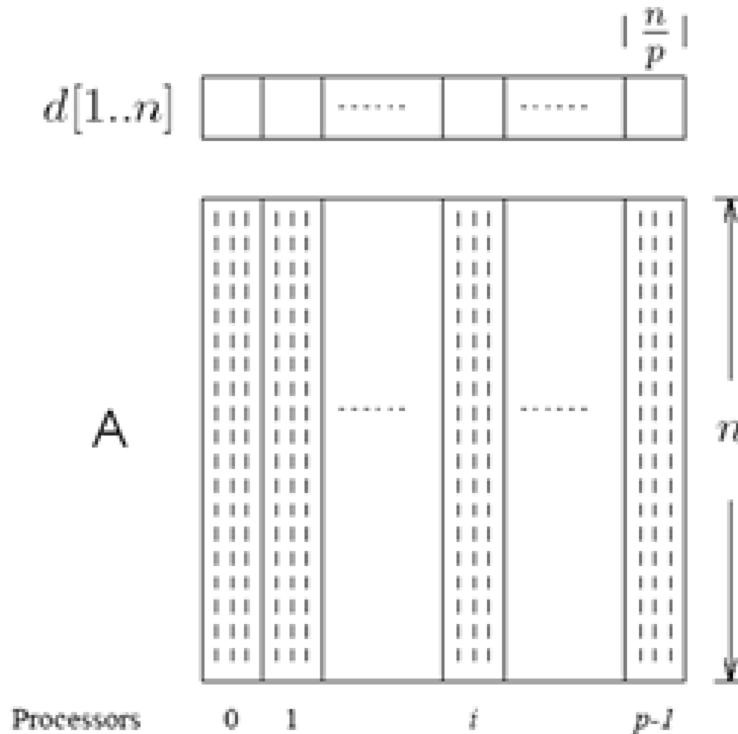
Ο ψευδοκώδικας για τον ακολουθιακό αλγόριθμο του Πριμ δίνεται παρακάτω:

```
1.  procedure PRIM_MST(V, E, w, r)
2.  begin
3.      V_T := {r};
4.      d[r] := 0;
5.      for all v ∈ (V − V_T) do
6.          if edge (r, v) exists set d[v] := w(r, v);
7.          else set d[v] := ∞;
8.      while V_T ≠ V do
9.      begin
10.         find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.         V_T := V_T ∪ {u};
12.         for all v ∈ (V − V_T) do
13.             d[v] := min{d[v], w(u, v)};
14.     endwhile
15. end PRIM_MST
```

11.2.1 Παράλληλη υλοποίηση του αλγόριθμου Prim

Θα περιγράψουμε την παράλληλη υλοποίηση του αλγόριθμου του Prim σε σύστημα κατανεμημένης μνήμης. Ο παράλληλος αλγόριθμος μπορεί να προσαρμοσθεί εύκολα και σε συστήματα διαμοιραζόμενης μνήμης. Οι επαναλήψεις του while-loop δεν μπορούν εύκολα να εκτελεστούν παράλληλα, αφού οι υπολογισμοί που εκτελούνται σε μία επανάληψη εξαρτώνται από τα αποτελέσματα των προηγουμένων επαναλήψεων. Αντίθετα οι επαναλήψεις του εσωτερικου for-loop είναι εύκολο να εκτελεστούν παράλληλα. Αν p είναι το πλήθος των διαθέσιμων διεργασιών και n το πλήθος των κορυφών του γραφήματος, οι κορυφές του γραφήματος χωρίζονται σε p υποσύνολα και κάθε διεργασία P_i αναλαμβάνει τις κορυφές $in/p, in/p+1, \ldots, (i+1)n/p-1$, όπου $i = 0, \ldots, p-1$.

Έστω V_i το σύνολο αυτών των κορυφών. Ο διαμοιρασμός αυτών των κορυφών στις διαθέσιμες διεργασίες συνεπάγεται και το χωρισμό του πίνακα γειτνίασης σε p κατακόρυφες λωρίδες πάχους n/p στηλών. Κάθε επεξεργαστής αναλαμβάνει την λωρίδα που αντιστοιχεί στις δικές του κορυφές. Με ανάλογο τρόπο χωρίζεται ο πίνακας d που αποθηκεύει τη τρέχουσα ελάχιστη απόσταση μίας κορυφής από το υπό κατασκευή ΕΓΔ. Ο χωρισμός του πίνακα γειτνίασης και του πίνακα d φαίνεται στο σχήμα που ακολουθεί.



Σε κάθε επανάληψη του while loop, κάθε διεργασία P_i υπολογίζει τοπικά την ελάχιστη απόσταση που χωρίζει τις κορυφές της από το υπο-κατασκευή ΕΓΔ, δηλαδή

$$d_i[u] = \min\{d_i[v] : v \in (V - V_T) \cap V_i\}.$$

Η πολυπλοκότητα αυτού του βήματος είναι $O(n/p)$. Το συνολικό ελάχιστο λαμβάνεται από όλα τα $d_i[u]$ χρησιμοποιώντας μία λειτουργία μείωσης (reduction) και αποθηκεύεται στη διεργασία P_0 (κόστος $O(\log p)$). Έτσι, η διεργασία P_0 έχει τη νέα κορυφή u που θα εισέλθει στο ΕΓΔ (σύνολο V_T).

Στη συνέχεια, η διεργασία P_0 στέλνει τη u σε όλες τις διεργασίες χρησιμοποιώντας τη λειτουργία της εκπομπής (κόστος $O(\log p)$).

Η διεργασία P_i, υπεύθυνη για την κορυφή u, σημειώνει τη u ότι ανήκει στο σύνολο V_T (κόστος $O(1)$).

Στο τέλος, κάθε διεργασία ενημερώνει τις τιμές των $d[v]$ των τοπικών της κορυφών (κόστος $O(n/p)$).

Η παραπάνω διαδικασία επαναλαμβάνεται, μέχρι όλες οι κορυφές να εισέλθουν στο σύνολο V_T. Η συνολική πολυπλοκότητα του αλγορίθμου θα είναι $O(n2/p + n \log p)$.

11.3 Εύρεση συντομότερων μονοπατιών με κοινή αφετηρία

Για ένα βεβαρημένο γράφημα $G = (V, E, w)$, το πρόβλημα των συντομότερων μονοπατιών με κοινή αφετηρία είναι η εύρεση των συντομότερων μονοπατιών από μία κορυφή $s \in V$ στις άλλες κορυφές του γραφήματος. Το πρόβλημα αυτό επιλύεται με τον αλγόριθμο του Dijkstra, ο οποίος παρουσιάζει πολλές ομοιότητες με τον αλγόριθμο του Prim. Συγκεκριμένα, σε κάθε βήμα του, ο αλγόριθμος συντηρεί το σύνολο V_T των κορυφών για τις οποίες είναι ήδη γνωστά τα συντομότερα μονοπάτια. Σε κάθε βήμα, το σύνολο V_T αυξάνεται κατά ένα κόμβο. Συγκεκριμένα, ο κόμβος αυτός είναι ο πιο κοντινός στη αφετηρία από όλους τους υπόλοιπους κόμβους που δεν ανήκουν στο σύνολο V_T. Σημαντική λεπτομέρεια του αλγορίθμου είναι ότι το μονοπάτι του

νεοεισερχόμενου κόμβου στο V_T θα πρέπει απαρτίζεται μόνο από κόμβους που είναι ήδη στο V_T. Ο ακολουθιακός κώδικας για τον αλγόριθμο του Dijkstra έχει ως εξής:

```
1.    procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2.    begin
3.        V_T := {s};
4.        for all v ∈ (V − V_T) do
5.            if (s, v) exists set l[v] := w(s, v);
6.            else set l[v] := ∞;
7.        while V_T ≠ V do
8.        begin
9.            find a vertex u such that l[u] := min{l[v]|v ∈ (V − V_T)};
10.           V_T := V_T ∪ {u};
11.           for all v ∈ (V − V_T) do
12.               l[v] := min{l[v], l[u] + w(u, v)};
13.       endwhile
14.   end DIJKSTRA_SINGLE_SOURCE_SP
```

11.3.1 Παράλληλη υλοποίηση του αλγόριθμου του Dijkstra

Η παράλληλη υλοποίηση του αλγόριθμου αυτού είναι παρόμοια με αυτή του αλγόριθμου του Prim. Τόσο ο πίνακας γειτνίασης όσο και ο πίνακας των αποστάσεων l μοιράζονται με τον ίδιο τρόπο στις διεργασίες. Κάθε διεργασία επιλέγει τοπικά την κορυφή που είναι κοντινότερη στην αφετηρία και στη συνέχεια, με μία λειτουργία μείωσης, βρίσκεται η συνολικά κοντινότερη κορυφή ως προς την αφετηρία. Στη συνέχεια, η κορυφή αυτή στέλνεται σε όλες τις διεργασίες με μία λειτουργία εκπομπής και οι διεργασίες ενημερώνουν το τμήμα του πίνακα l που κατέχουν. Είναι εύκολο να δει κανείς ότι η πολυπλοκότητα του παράλληλου αλγόριθμου Dijkstra είναι ίδια με αυτή του αλγορίθμου Prim.

11.4 Εύρεση συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών

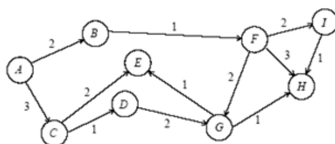
Δοθέντος ενός βεβαρημένου γραφήματος $G = (V, E, w)$, το πρόβλημα των συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών είναι η εύρεση των συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών $v_i, v_j \in V$.

Ο πρώτος αλγόριθμος που επιλύει το πρόβλημα βασίζεται στον πολλαπλασιασμό πινάκων. Συγκεκριμένα αν υπολογίσουμε το τετράγωνο του πίνακα γειτνίασης A, δηλαδή τον $A^2 = A \cdot A$, όπου στη θέση του πολλαπλασιασμού εκτελείται πρόσθεση και στη θέση της πρόσθεσης τώρα εκτελείται η λειτουργία του ελάχιστου, τότε το αποτέλεσμα A^2 θα περιέχει όλα τα συντομότερα μονοπάτια που έχουν μήκος το πολύ 2. Συγκεκριμένα, εκτελούμε τον εξής μετασχηματισμό:

$$a_{i,j} = \sum_{k=0}^{|V|-1} a_{i,k}a_{k,j} \rightarrow a_{i,j} = \min\{a_{i,j}, a_{i,0} + a_{0,j}, a_{i,1} + a_{1,j}, \ldots, a_{i,|V|-1} + a_{|V|-1,j}\}.$$

Γενικεύοντας αυτή την παρατήρηση, μπορούμε να δούμε ότι ο πίνακας A^n περιέχει τα κόστη των συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών. Ο πίνακας A^n υπολογίζεται με διπλασιασμό δυνάμεων, δηλαδή A, A^2, A^4, A^8, κοκ. Χρειαζόμαστε $\log n$ πολλαπλασιασμούς πινάκων. Ο ακολουθιακός υπολογισμός για τον πολλαπλασιασμό πινάκων απαιτεί χρόνο $O(n^3)$. Άρα η συνολική ακολουθιακή πολυπλοκότητα θα είναι $O(n^3 \log n)$. Αυτός ο αλγόριθμος δεν είναι βέλτιστος, αφού ο καλύτερος ακολουθιακός αλγόριθμος έχει πολυπλοκότητα $O(n^3)$.

Η τεχνική αυτή μπορεί να υλοποιηθεί παράλληλα, με την παράλληλη εκτέλεση καθενός από του $O(\log n)$ πολλαπλασιασμών που απαιτούνται συνολικά.



$$A^1 = \begin{pmatrix} 0 & 2 & 3 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & 2 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^2 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & \infty & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^8 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Σχήμα 11.2: Παράδειγμα εκτέλεσης της τεχνικής του πολλαπλασιασμού πινάκων.

11.5 Αλγόριθμος του Floyd

Ο αλγόριθμος αυτός επιλύει το πρόβλημα των συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών και βασίζεται στην εξής παρατήρηση:

Για οποιοδήποτε ζεύγος κορυφών $v_i, v_j \in V$, έστω $p_{i,j}^{(k)}$ (βάρους $d_{i,j}^{(k)}$) το μονοπάτι με το ελάχιστο βάρος μεταξύ όλων των μονοπατιών από τη κορυφή v_i στη κορυφή v_j, των οποίων οι ενδιάμεσες κορυφές ανήκουν στο σύνολο $\{v_1, v_2, \ldots, v_k\}$. Αν η κορυφή v_k δεν είναι στο συντομότερο μονοπάτι από την κορυφή v_i στη κορυφή v_j, τότε το μονοπάτι $p_{i,j}^{(k)}$ είναι το ίδιο με $p_{i,j}^{(k-1)}$. Αν v_k είναι στο $p_{i,j}^{(k)}$, τότε μπορούμε να σπάσουμε το μονοπάτι $p_{i,j}^{(k)}$ σε δύο μονοπάτια • ένα από την κορυφή v_i στην κορυφή v_k και ένα από τη v_k στη v_j. Κάθε ένα από αυτά τα μονοπάτια χρησιμοποιεί κορυφές από το σύνολο $\{v1, v2, \ldots, v_{k-1}\}$. Από τις παραπάνω παρατηρήσεις, η ακόλουθη αναδρομική εξίσωση προκύπτει:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j), & k = 0, \\ \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\}, & k > 0. \end{cases}$$

Επομένως, η τιμή $d_{i,j}^{(n)}$ θα δίνει το κόστος του συντομότερου μονοπατιού μεταξύ των κορυφών v_i και v_j.

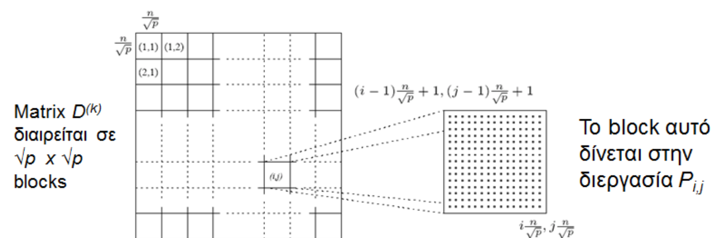
Με βάση την προηγούμενη αναδρομική σχέση, ο ακολουθιακός αλγόριθμος του Floyd θα έχει ως εξής:

```
1.    procedure FLOYD_ALL_PAIRS_SP(A)
2.    begin
3.        D^(0) = A;
4.        for k := 1 to n do
5.            for i := 1 to n do
6.                for j := 1 to n do
7.                    d_{i,j}^(k) := min( d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1) );
8.    end FLOYD_ALL_PAIRS_SP
```

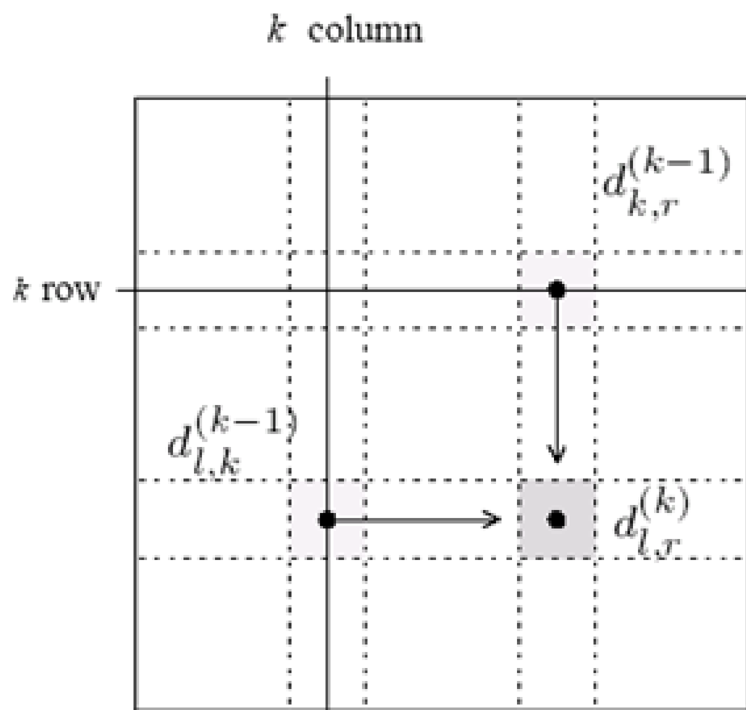
Η πολυπλοκότητα του αλγορίθμου είναι $O(n^3)$.

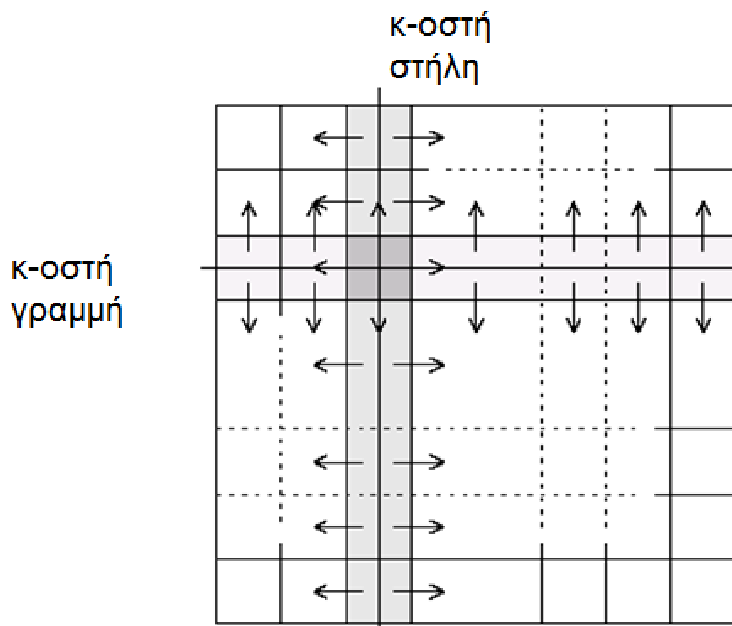
11.5.1 Παράλληλη υλοποίηση του αλγόριθμου του Floyd

Ο πίνακας $D^{(k)}$ διαιρείται σε p blocks μεγέθους $(n/\sqrt{p}) \times (n/\sqrt{p})$.



Κάθε διεργασία αναλαμβάνει ένα από αυτά τα blocks. Σε κάθε επανάληψη, κάθε διεργασία ενημερώνει το τμήμα του πίνακα που έχει αναλάβει. Για να υπολογίσει το $d_{l,r}^{(k-1)}$, η διεργασία $P_{i,j}$ πρέπει να λάβει τα στοιχεία $d_{l,k}^{(k-1)}$ και $d_{k,r}^{(k-1)}$.





Γενικά, κατά την k-οστή επανάληψη, κάθε μία από τις $\sqrt{p}$ διεργασίες οι οποίες περιέχουν τμήμα της k-οστής γραμμής στέλνουν το τμήμα αυτό στις $\sqrt{p} - 1$ διεργασίες της ίδιας στήλης. Όμοια, κάθε μία από τις $\sqrt{p}$ διεργασίες οι οποίες περιέχουν τμήμα της k-οστής στήλης στέλνουν το τμήμα τους στις $\sqrt{p}$"1 διεργασίες στην ίδια γραμμή.

Έτσι, ο παράλληλος αλγόριθμος για το αλγόριθμο του Floyd έχει ως εξής:

```
1.   procedure FLOYD_2DBLOCK(D^(0))
2.   begin
3.       for k := 1 to n do
4.       begin
5.           each process P_{i,j} that has a segment of the k^th row of D^(k-1);
                 broadcasts it to the P_{*,j} processes;
6.           each process P_{i,j} that has a segment of the k^th column of D^(k-1);
                 broadcasts it to the P_{i,*} processes;
7.           each process waits to receive the needed segments;
8.           each process P_{i,j} computes its part of the D^(k) matrix;
9.       end
10.  end FLOYD_2DBLOCK
```

Στον ψευδοκώδικα, $P_{*,j}$ είναι όλες οι διεργασίες της j-οστής στήλης και $P_{i,*}$ είναι όλες οι διεργασίες στην i-οστη γραμμή. Ο πίνακας $D^{(0)}$ είναι ο αρχικός πίνακας γειτνίασης.

Ανάλυση πολυπλοκότητας αλγορίθμου: Σε κάθε επανάληψη του αλγορίθμου, οι διεργασίες της k-οστής γραμμής και της k-οστής στήλης εκτελούν λειτουργία εκπομπής κατά μήκος των στηλών και των γραμμών τους αντίστοιχα. Το πλήθος των δεδομένων που στέλνεται σε κάθε λειτουργία εκπομπής είναι $n/\sqrt{p}$. Σε ένα σύστημα κατανεμημένης μνήμης, η εκπομπή ενός στοιχείου μπορεί να ολοκληρωθεί σε χρόνο $\Theta(\log p)$. Άρα, συνολικά ο χρόνος για αυτό το βήμα θα είναι $\Theta((n \log p)/\sqrt{p})$. Το βήμα συγχρονισμού μετά από κάθε επανάληψη απαιτεί χρόνο $\Theta(\log p)$. Σε κάθε επανάληψη, ο χρόνος που απαιτείται για τους υπολογισμούς του αλγορίθμου είναι $\Theta(n^2/p)$. Συνολικά, ο χρόνος εκτέλεσης του αλγορίθμου θα είναι:

$$T_p = \Theta\left(\frac{n^3}{p}\right) + \Theta\left(\frac{n^2}{\sqrt{p}} \log p\right).$$

11.6 Μεταβατική κλειστότητα

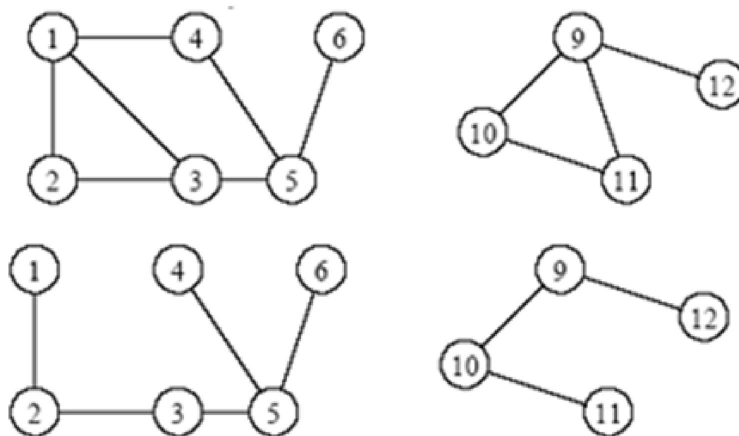
Αν $G = (V, E)$ είναι ένα γράφημα, τότε η μεταβατική κλειστότητα του G ορίζεται ως το γράφημα $G^* = (V, E^*)$, όπου

$$E^* = \{(v_i, v_j) : \text{ υπάρχει μονοπάτι από τη κορυφή } v_i \text{ στη κορυφή } v_j \text{ στο } G\}.$$

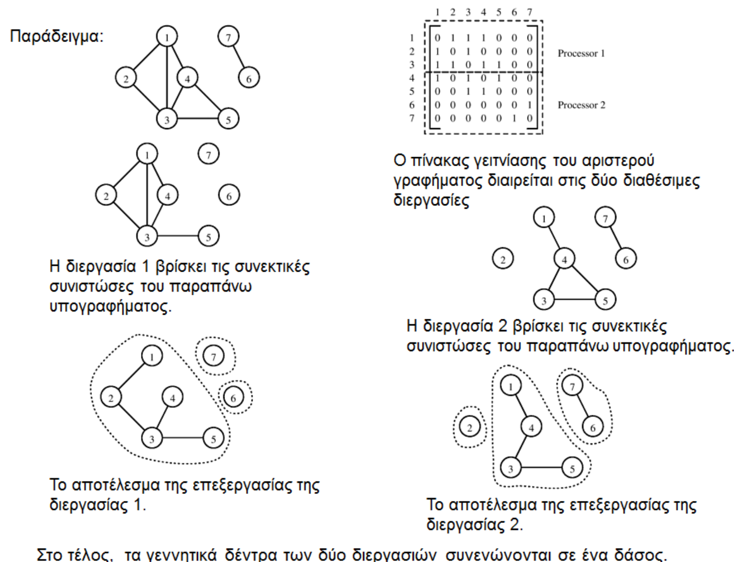
Ο πίνακας συνεκτικότητας του G είναι ένας πίνακας $A^* = (a^*_{i,j})$, όπου το στοιχείο $a^*_{i,j} = 1$ αν και μόνο αν υπάρχει μονοπάτι από την κορυφή v_i στην κορυφή v_j ή όταν $i = j$. Σε διαφορετική περίπτωση, $a^*_{i,j} = \infty$. Για να υπολογίσουμε τον A^*, αρχικά θεωρούμε το βάρος κάθε ακμής ίσο με 1 και στη συνέχεια μπορούμε να εφαρμόσουμε σε αυτό το βεβαρημένο γράφημα οποιοδήποτε αλγόριθμο που επιλύει το πρόβλημα συντομότερων διαδρομών μεταξύ όλων των ζευγών κορυφών. Οι συνεκτικές συνιστώσες ενός μη κατευθυνόμενου γραφήματος είναι κλάσεις ισοδυναμίας της σχέσης "έχει πρόσβαση στη" στο σύνολο των κορυφών του γραφήματος.

11.7 Εύρεση συνεκτικών συνιστωσών: Αναζήτηση πρώτα κατά βάθος

Η εκτέλεση της αναζήτησης πρώτα κατά βάθος σε ένα γράφημα δημιουργεί ένα δάσος • κάθε δέντρο στο δάσος αντιστοιχεί σε μία διαφορετική συνεκτική συνιστώσα. Για παράδειγμα, με εφαρμογή της αναζήτησης πρώτα κατά βάθος στο γράφημα που ακολουθεί, προκύπτουν δύο διαφορετικά δέντρα τα οποία αντιστοιχούν στις δύο συνεκτικές συνιστώσες του γραφήματος.



Για την παράλληλη υλοποίηση, το αρχικό γράφημα μοιράζεται στις διαθέσιμες διεργασίες και κάθε διεργασία υπολογίζει τις συνεκτικές συνιστώσες αποκλειστικά στο τμήμα του γραφήματος που της αντιστοιχεί. Σε αυτό το σημείο, έχουν δημιουργηθεί p γεννητικά δάση. Στο δεύτερο βήμα, τα γεννητικά δάση συγχωνεύονται σε ζευγάρια μέχρι ένα γεννητικό δάσος να απομείνει.



Για τη συγχώνευση των γεννητικών δασών, ο αλγόριθμος χρησιμοποιεί τα σύνολα ακμών των δασών A και B, τα οποία είναι ξένα μεταξύ τους. Ορίζουμε τις ακόλουθες λειτουργίες στα ξένα σύνολα:

- find(x). Επιστρέφει ένα δείκτη στο στοιχείο αντιπρόσωπο του συνόλου που περιέχει το x. Κάθε σύνολο έχει το δικό του μοναδικό αντιπρόσωπο.
- union(x, y). Ενώνει δύο σύνολα τα οποία περιέχουν τα στοιχεία x και y. Γίνεται η υπόθεση ότι τα δύο σύνολα είναι ξένα πριν την εφαρμογή της λειτουργίας.

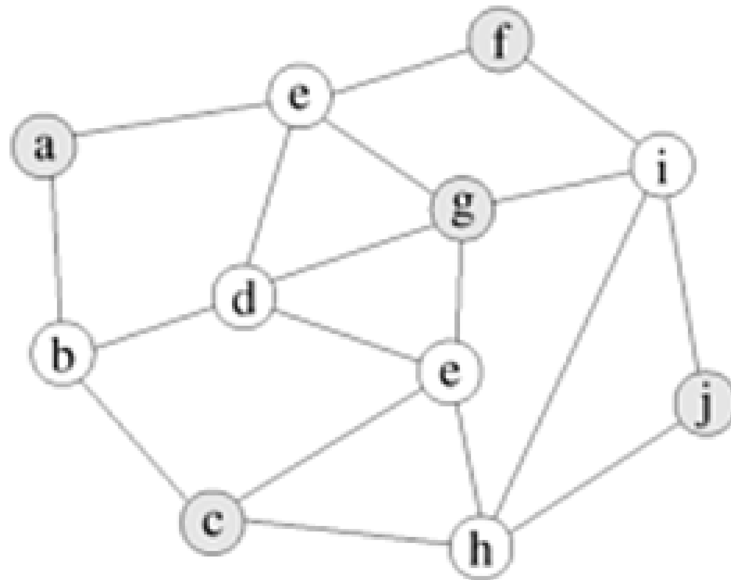
Για την συγχώνευση του δάσους A στο δάσος B, για κάθε ακμή (u, v) του A, εκτελείται μία find λειτουργία, για να προσδιορισθεί αν οι κορυφές της ακμής είναι και οι δύο στο ίδιο δέντρο του B. Αν αυτό ισχύει, δεν απαιτείται καμία λειτουργία union. Στην αντίθετη περίπτωση, τα δύο δέντρα του B που περιέχουν τις κορυφές u και v ενώνονται με μία λειτουργία union. Συνολικά, η συγχώνευση των δασών A και B απαιτεί το πολύ $2(n-1)$ λειτουργίες find και $(n-1)$ λειτουργίες union.

Στη γενική περίπτωση, όταν υπάρχουν p διαθέσιμες διεργασίες, ο πίνακας γειτνίασης $n \times n$ διαιρείται σε p blocks. Κάθε επεξεργαστής μπορεί να υπολογίσει το τοπικό του γεννητικό δάσος σε χρόνο $\Theta(n^2/p)$. Η συγχώνευση των δασών μπορεί να γίνει οργανώνοντας τις διαθέσιμες διεργασίες σε ένα λογικό δέντρο. Υπάρχουν $\log p$ στάδια συγχώνευσης καθένα από τα οποία απαιτεί χρόνο $\Theta(n)$. Έτσι, το συνολικό κόστος λόγω συγχώνευσης είναι $\Theta(n \log p)$. Κατά τη διάρκεια κάθε βήματος συγχώνευσης, τα γεννητικά δάση στέλνονται στους πιο κοντινούς γείτονες. Συνολικά, $\Theta(n)$ ακμές γεννητικών ακμών μεταδίδονται. Έτσι, το συνολικό κόστος επικοινωνίας θα είναι $\Theta(n \log p)$. Έτσι, το συνολικό κόστος του αλγορίθμου θα είναι:

$$T_p = \Theta\left(\frac{n^2}{p}\right) + \Theta(n \log p).$$

11.8 Εύρεση ενός Μέγιστου (Maximal) Ανεξάρτητου Συνόλου

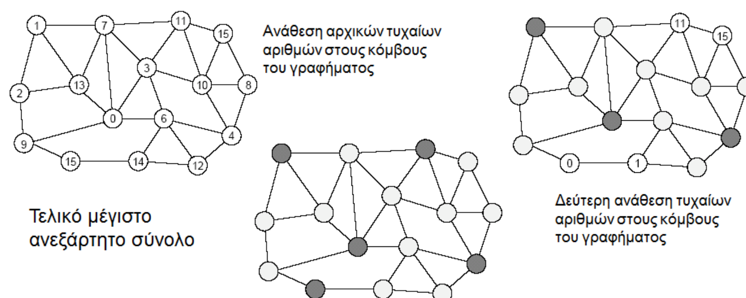
Ένα σύνολο κορυφών $I \subset V$ ονομάζεται ανεξάρτητο, αν κανένα ζεύγος κορυφών στο I δεν συνδέεται με μία ακμή στο G. Ένα ανεξάρτητο σύνολο λέγεται μέγιστο (maximal) αν με τη συμπερίληψη οποιασδήποτε άλλης κορυφής που δεν ανήκει στο σύνολο I, η ιδιότητα της ανεξαρτησίας παραβιάζεται.



Στο συγκεκριμένο παράδειγμα, το σύνολο των κορυφών $\{a, d, i, h\}$ είναι ανεξάρτητο. Τα σύνολα $\{a, c, j, f, g\}$ και $\{a, d, h, f\}$ είναι μέγιστα ανεξάρτητα σύνολα.

Ένας απλός αλγόριθμος για την εύρεση του μέγιστου ανεξάρτητου συνόλου αρχίζει με το ανεξάρτητο σύνολο I αρχικά κενό. Επίσης, υπάρχει ένα σύνολο C που περιέχει τις υποψήφιες κορυφές προς εισαγωγή στο ανεξάρτητο σύνολο I και αρχικά περιλαμβάνει όλες τις κορυφές του γραφήματος. Σε κάθε βήμα, μία κορυφή v επιλέγεται από το C η οποία στη συνέχεια μετακινείται στο σύνολο I, ενώ όλες οι κορυφές που είναι γειτονικές στην v αφαιρούνται από το σύνολο C. Αυτή η διαδικασία επαναλαμβάνεται, μέχρι το σύνολο C να αδειάσει. Το πρόβλημα με την παραπάνω διαδικασία είναι ότι είναι εγγενώς ακολουθιακή.

Ο παράλληλος αλγόριθμος για την εύρεση του μέγιστου ανεξάρτητου συνόλου εισάγει τυχαιότητα στην επιλογή της επόμενης κορυφής που θα συμπεριληφθεί στο ανεξάρτητο σύνολο προκειμένου να είναι δυνατή η παράλληλη εκτέλεση κάποιων ενεργειών. Ο αλγόριθμος βασίζεται στον αλγόριθμο του Luby για το χρωματισμό γραφήματος. Αρχικά, κάθε κόμβος είναι στο σύνολο C των υποψήφιων προς εισαγωγή κόμβων. Κάθε κόμβος παράγει ένα μοναδικό τυχαίο αριθμό και το στέλνει στους γείτονες του. Αν ο αριθμός ενός κόμβου είναι μεγαλύτερος από αυτούς των γειτόνων του, τότε ο κόμβος αυτός συμπεριλαμβάνεται στο σύνολο I και αφαιρείται από το σύνολο C. Η διαδικασία συνεχίζεται, μέχρι το σύνολο C να γίνει κενό. Κατά μέσο όρο, αυτός ο αλγόριθμος συγκλίνει μετά από $O(\log |V|)$ τέτοια βήματα. Ακολουθεί ένα παράδειγμα εφαρμογής του αλγορίθμου. Με γκρι χρωματίζονται οι κόμβοι που ανήκουν στο ανεξάρτητο σύνολο, ενώ οι κενοί κόμβοι είναι γείτονες αυτών των κόμβων.



Για την παράλληλη εκτέλεση του αλγορίθμου, απαιτούνται τρεις πίνακες n στοιχείων. Συγκεκριμένα, ο πίνακας I που αποθηκεύει κόμβους του μέγιστου ανεξάρτητου συνόλου, ο C που

αποθηκεύει το σύνολο των υποψήφιων κορυφών και το σύνολο R που αποθηκεύει τους τυχαίους αριθμούς. Τα στοιχεία των πινάκων I και C παίρνουν δυαδικές τιμές, με το 1 να δείχνει ότι ο αντίστοιχος κόμβος ανήκει στο σύνολο και το 0 να δείχνει το αντίθετο.

Στη συνέχεια, μοιράζουμε τον πίνακα C στις p διεργασίες. Κάθε διεργασία παράγει τις τυχαίες τιμές των κόμβων που έχει αναλάβει και τις αποθηκεύει στις αντίστοιχες θέσεις του πίνακα R. Με βάση τις τιμές του πίνακα R, κάθε διεργασία υπολογίζει ποιες είναι οι επόμενες κορυφές μεταξύ των κορυφών που έχει αναλάβει που θα μπουν στο μέγιστο ανεξάρτητο σύνολο.

Ο πίνακας C ενημερώνεται θέτοντας 0 στις θέσεις των κόμβων που εισήλθαν στο μέγιστο ανεξάρτητο σύνολο ή είναι γείτονες των κόμβων που μόλις εισήλθαν στο ανεξάρτητο σύνολο. Επίσης, θέτουμε 1 στις θέσεις του πίνακα I που αντιστοιχούν στους νεοεισερχόμενους κόμβους. Η συνολική πολυπλοκότητα του αλγορίθμου εξαρτάται από τη συγκεκριμένη δομή του γραφήματος.

11.9 Συντομότερα μονοπάτια με κοινή αφετηρία

Ο αλγόριθμος του Johnson αποτελεί τροποποίηση του αλγόριθμου του Dijkstra, ώστε να εκτελείται αποδοτικά στη περίπτωση των αραιών γραφημάτων. Η τροποποίηση αυτή λαμβάνει υπόψη το γεγονός ότι το βήμα ελαχιστοποίησης του αλγόριθμου Dijkstra χρειάζεται, στην πράξη, να εκτελείται μόνο στους κόμβους οι οποίοι είναι γειτονικοί ως προς κόμβο του οποίου μόλις προσδιορίσθηκε η συντομότερη διαδρομή από το αλγόριθμο Dijkstra. Ο αλγόριθμος του Johnson χρησιμοποιεί μία ουρά προτεραιότητας Q για να αποθηκεύσει την τιμή $l[v]$, για κάθε κορυφή $v \in (V"V_T)$. Ο ακολουθιακός αλγόριθμος έχει ως εξής:

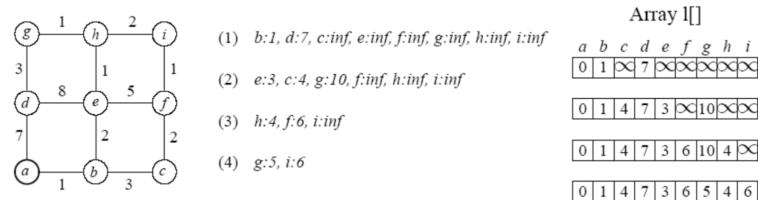
```
1.   procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
2.   begin
3.       Q := V;
4.       for all v ∈ Q do
5.           l[v] := ∞;
6.       l[s] := 0;
7.       while Q ≠ ∅ do
8.       begin
9.           u := extract_min(Q);
10.          for each v ∈ Adj[u] do
11.              if v ∈ Q and l[u] + w(u, v) < l[v] then
12.                  l[v] := l[u] + w(u, v);
13.      endwhile
14.  end JOHNSON_SINGLE_SOURCE_SP
```

Πιστή εκτέλεση του αλγόριθμου του Johnson δεν επιτρέπει την εκτέλεση πολλών λειτουργιών παράλληλα. Αυτό οφείλεται στο γεγονός ότι ο ακολουθιακός αλγόριθμος επιβάλλει μία συγκεκριμένη σειρά επίσκεψης των κορυφών του γραφήματος. Για να αυξηθεί ο παραλληλισμός στο συγκεκριμένο υπολογισμό, χρειάζεται να εξεταστούν πολλοί κόμβοι ταυτόχρονα. Για το σκοπό αυτό, εξάγονται p κορυφές (όσες και οι διαθέσιμες διεργασίες) από την ουρά προτεραιότητας και επεκτείνονται ανάλογα τα συντομότερα μονοπάτια. Στη συνέχεια, ενημερώνονται τα κόστη όλων των γειτόνων αυτών των κορυφών. Αν στη συνέχεια διαπιστωθεί ότι υπάρχει συντομότερο μονοπάτι για ένα κόμβο που βγήκε από την ουρά προτεραιότητας, ο κόμβος αυτός εισάγεται ξανά στην ουρά προτεραιότητας.

Στο παράδειγμα που ακολουθεί, υπάρχουν τρεις διεργασίες και το ζητούμενο είναι η εύρεση των συντομότερων διαδρομών με αρχή την κορυφή a. Αρχικά, οι διεργασίες P_0 και P_1 εξάγουν τις κορυφές b και d και στη συνέχεια ενημερώνουν τις τιμές l των κορυφών που είναι γειτονικές

στις b και d. Στο δεύτερο βήμα, οι διεργασίες P_0, P_1 και P_2 εξάγουν τις κορυφές e, c και g και στη συνέχεια ενημερώνουν τις τιμές l των κορυφών που είναι γειτονικές σε αυτές.

Παρατηρείστε ότι όταν η διεργασία P_0 εξάγει την κορυφή h, διαπιστώνει ότι $l[h]+w(h,g) = 5$ σε σύγκριση με την τιμή $l[g] = 10$ η οποία εξήχθη από την ουρά προτεραιότητας στη προηγούμενη επανάληψη. Αυτό έχει ως αποτέλεσμα, η κορυφή g να επιστρέψει στην ουρά προτεραιότητας με ενημερωμένη την τιμή $l[g]$.



Αν υλοποιήσουμε την παραπάνω τεχνική σε σύστημα κατανεμημένης μνήμης, η διαχείριση της κοινής ουράς αποτελεί αιτία καθυστέρησης του συνολικού υπολογισμού. Για αυτό το λόγο, πολλαπλές ανεξάρτητες ουρές χρησιμοποιούνται, μία για κάθε διεργασία. Συγκεκριμένα, κάθε διεργασία χτίζει τη δική της ουρά προτεραιότητας, η οποία περιέχει τις κορυφές που χειρίζεται η διεργασία. Όταν μία διεργασία P_i εξάγει την κορυφή $u \in V_i$, στέλνει μήνυμα με την τιμή $l[u]$ σε όλες τις διεργασίες που χειρίζονται κορυφές γειτονικές στην κορυφή u.

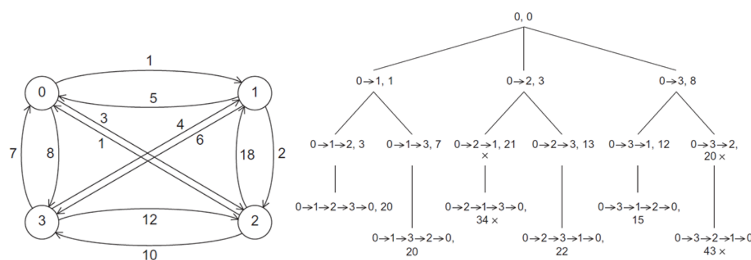
Κάθε διεργασία P_j, αφού λάβει το μήνυμα από την P_i, θέτει την τιμή $l[v]$ στην τιμή $\min\{l[v], l[u]+w(u,v)\}$. Αν ανακαλυφθεί συντομότερο μονοπάτι για τον κόμβο v, η τιμή $l[v]$ επανεισάγεται στην τοπική ουρά προτεραιότητας. Ο αλγόριθμος τερματίζει μόνο όταν όλες οι ουρές αδειάσουν.

Μπορούμε να χρησιμοποιήσουμε διαφορετικούς τρόπους διαμοίρασης των κορυφών του γραφήματος στις διάφορες διεργασίες, ανάλογα με τη συγκεκριμένη δομή του γραφήματος που επεξεργαζόμαστε.

11.10 Το Πρόβλημα του Περιοδεύοντος Πωλητή (ΠΠΠ)

Το πρόβλημα αυτό είναι ένα NP-ςομπλετε πρόβλημα. Δεν υπάρχει γνωστή λύση για το συγκεκριμένο πρόβλημα η οποία να είναι καλύτερη σε όλες τις περιπτώσεις σε σχέση με την εξαντλητική αναζήτηση.

Ένα παράδειγμα με 4 πόλεις:



Χρόνοι εκτέλεσης των τριών ακολουθιακών υλοποιήσεων της τεχνικής της δενδρικής αναζήτησης (σε δευτερόλεπτα):

Recursive	First iterative	Second iterative
30.5	29.2	32.9

Το γράφημα περιέχει 15 πόλεις. Και οι τρεις υλοποιήσεις επισκέπτονται περίπου 95.000.000 κόμβους δέντρου.

```
void Depth_first_search(tour_t tour) {
   city_t city;

   if (City_count(tour) == n) {
      if (Best_tour(tour))
         Update_best_tour(tour);
   } else {
      for each neighboring city
         if (Feasible(tour, city)) {
            Add_city(tour, city);
            Depth_first_search(tour);
            Remove_last_city(tour);
         }
   }
}  /* Depth_first_search */
```

Σχήμα 11.3: Ψευδοκώδικας για την αναδρομική λύση του ΠΠΠ χρησιμοποιώντας την αναζήτηση πρώτα κατά βάθος.

11.10.1 Υπολογισμός του κόστους του τρέχοντος καλύτερου γύρου πόλεων

Όταν μία διεργασία ολοκληρώσει ένα γύρο, χρειάζεται να ελέγξει αν έχει την καλύτερη λύση που έχει υπολογισθεί μέχρι εκείνη τη στιγμή. Η συνάρτηση global Best_tour διαβάζει μόνο το συνολικά καλύτερο κόστος, και έτσι δεν χρειάζεται να χρησιμοποιήσουμε μηχανισμό κλειδώματος για να εξασφαλίζουμε την ασφαλή πρόσβαση στη μεταβλητή, αφού δεν υπάρχει ανταγωνισμός μεταξύ των αναγνωστών. Αν η διεργασία δεν έχει καλύτερη λύση, τότε δεν ενημερώνει την μεταβλητή Best_tour. Σε περίπτωση που μία άλλη διεργασία ενημερώνει τη μεταβλητή την ίδια χρονική στιγμή, η διεργασία αναγνώστης μπορεί να δει την παλαιά ή τη νέα τιμή.

Θα ήταν προτιμότερο μία διεργασία να διαβάζει πάντα την πιο πρόσφατη τιμή, άλλα αυτή η εγγύηση έχει σχετικά υψηλό κόστος. Στην περίπτωση που μία διεργασία ελέγξει και αποφασίσει ότι έχει καλύτερη συνολικά λύση, χρειάζεται να εξασφαλίσουμε δύο πράγματα:

1. Ότι η διαδικασία κλειδώνει την τιμή Best_tour με μία μεταβλητή mutex, αποτρέποντας τον ανταγωνισμό από τις άλλες διεργασίες.

2. Στην περίπτωση που ο πρώτος έλεγχος ήταν ως προς μία παλαιά τιμή, ενώ εντωμεταξύ κάποια άλλη διεργασία έχει ενημερώσει τη μεταβλητή, δεν θέλουμε να θέσουμε στη μεταβλητή μία χειρότερη τιμή σε σχέση με τη νεότερη που έχει γραφτεί.

Χειριζόμαστε αυτές τις περιπτώσεις με μηχανισμό κλειδώματος και επανέλεγχο της μεταβλητής.

11.10.2 Δυναμική παραλληλοποίηση της αναζήτησης δέντρου χρησιμοποιώντας Pthreads

Υπάρχουν θέματα σωστού τερματισμού των νημάτων. Ο κώδικας που εκτελείται από ένα νήμα πριν μοιράσει την εργασία του έχει ως εξής:

- Ελέγχει ότι έχει τουλάχιστον δύο γύρους πόλεων στη στοίβα του.

- Ελέγχει ότι υπάρχουν νήματα που περιμένουν να λάβουν νέα εργασία.

```
for (city = n-1; city >= 1; city--)
   Push(stack, city);
while (!Empty(stack)) {
   city = Pop(stack);
   if (city == NO_CITY) // End of child list, back up
      Remove_last_city(curr_tour);
   else {
      Add_city(curr_tour, city);
      if (City_count(curr_tour) == n) {
         if (Best_tour(curr_tour))
            Update_best_tour(curr_tour);
         Remove_last_city(curr_tour);
      } else {
         Push(stack, NO_CITY);
         for (nbr = n-1; nbr >= 1; nbr--)
            if (Feasible(curr_tour, nbr))
               Push(stack, nbr);
      }
   } /* if Feasible */
} /* while !Empty */
```

Σχήμα 11.4: Ψευδοκώδικας για την υλοποίηση της αναζήτησης πρώτα κατά βάθος για το ΠΠΠ χωρίς αναδρομή.

- Ελέγχει αν η μεταβλητή new_stack έχει τιμή NULL.

Χρόνοι εκτέλεσης των προγραμμάτων με βάση το πρότυπο Pthreads (σε δευτερόλεπτα):

Προβλήματα με 15 πόλεις

Threads	First Problem			Second Problem		
	Serial	Static	Dynamic	Serial	Static	Dynamic
1	32.9	32.7	34.7 (0)	26.0	25.8	27.5 (0)
2		27.9	28.9 (7)		25.8	19.2 (6)
4		25.7	25.9 (47)		25.8	9.3 (49)
8		23.8	22.4 (180)		24.0	5.7 (256)

πλήθος χωρισμών
των στοιβών

11.11 Παράλληλη υλοποίηση της αναζήτησης δέντρου με OpenMP

Τα ίδια θέματα προκύπτουν κατά τη στατική ή δυναμική παράλληλη υλοποίηση της αναζήτησης δέντρου με OpenMP. Χρειάζονται μόνο κάποιες μικρές αλλαγές.

Pthreads
```
if (my_rank == whatever)
```
```
#  pragma omp single
```
OpenMP

Η απόδοση των υλοποιήσεων αναζήτησης δέντρου με OpenMP και Pthreads (σε δευτερόλεπτα):

```
Push_copy(stack, tour);  // Tour that visits only the hometown
while (!Empty(stack)) {
   curr_tour = Pop(stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour))
         Update_best_tour(curr_tour);
   } else {
      for (nbr = n-1; nbr >= 1; nbr--)
         if (Feasible(curr_tour, nbr)) {
            Add_city(curr_tour, nbr);
            Push_copy(stack, curr_tour);
            Remove_last_city(curr_tour);
         }
   }
   Free_tour(curr_tour);
}
```

Σχήμα 11.5: Ψευδοκώδικας για τη δεύτερη λύση στο ΠΠΠ η οποία δεν χρησιμοποιεί αναδρομή.

```
/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
   return tour->cities[i];
}  /* Tour_city */
```

```
/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
```

Σχήμα 11.6: Χρησιμοποίηση macros προεπεξεργασίας.

	First Problem						Second Problem					
	Static		Dynamic				Static		Dynamic			
Th	OMP	Pth	OMP		Pth		OMP	Pth	OMP		Pth	
1	32.5	32.7	33.7	(0)	34.7	(0)	25.6	25.8	26.6	(0)	27.5	(0)
2	27.7	27.9	28.0	(6)	28.9	(7)	25.6	25.8	18.8	(9)	19.2	(6)
4	25.4	25.7	33.1	(75)	25.9	(47)	25.6	25.8	9.8	(52)	9.3	(49)
8	28.0	23.8	19.2	(134)	22.4	(180)	23.8	24.0	6.3	(163)	5.7	(256)

11.12 Υλοποίηση της αναζήτησης δέντρου χρησιμοποιώντας MPI και στατική διαμέριση

Το MPI παρέχει τέσσερις τύπους για την εντολή αποστολής μηνύματος:

- Standard
- Synchronous
- Ready
- Buffered

Απόδοση υλοποιήσεων MPI και Pthreads (σε δευτερόλεπτα):

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

Σχήμα 11.7: Ψευδοκώδικας για την στατικά παράλληλη λύση του ΠΠΠ με Pthreads.

```
if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
      new_stack == NULL) {
   lock term_mutex;
   if (threads_in_cond_wait > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      pthread_cond_signal(&term_cond_var);
   }
   unlock term_mutex;
   return 0;  /* Terminated = False; don't quit */
} else if (!Empty(my_stack)) { /* Stack not empty, keep working */
   return 0;  /* Terminated = false; don't quit */
} else { /* My stack is empty */
   lock term_mutex;
   if (threads_in_cond_wait == thread_count-1) { /* Last thread   */
                                                 /* running       */
      threads_in_cond_wait++;
      pthread_cond_broadcast(&term_cond_var);
      unlock term_mutex;
      return 1;  /* Terminated = true; quit */
```

Σχήμα 11.8: Ψευδοκώδικας για συνάρτηση Terminated - Υλοποίηση με Pthreads.

Th/Pr	First Problem				Second Problem			
	Static		Dynamic		Static		Dynamic	
	Pth	MPI	Pth	MPI	Pth	MPI	Pth	MPI
1	35.8	40.9	41.9 (0)	56.5 (0)	27.4	31.5	32.3 (0)	43.8 (0)
2	29.9	34.9	34.3 (9)	55.6 (5)	27.4	31.5	22.0 (8)	37.4 (9)
4	27.2	31.7	30.2 (55)	52.6 (85)	27.4	31.5	10.7 (44)	21.8 (76)
8		35.7		45.5 (165)		35.7		16.5 (161)
16		20.1		10.5 (441)		17.8		0.1 (173)

```
      } else { /* Other threads still working, wait for work */
         threads_in_cond_wait++;
         while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
         /* We've been awakened */
         if (threads_in_cond_wait < thread_count) { /* We got work */
            my_stack = new_stack;
            new_stack = NULL;
            threads_in_cond_wait--;
            unlock term_mutex;
            return 0;  /* Terminated = false */
         } else {  /* All threads done */
            unlock term_mutex;
            return 1;  /* Terminated = true; quit */
         }
      }  /* else wait for work */
   }  /* else my_stack is empty */
```

Σχήμα 11.9: Ψευδοκώδικας για συνάρτηση Terminated.

```
typedef struct {
   my_stack_t new_stack;
   int threads_in_cond_wait;
   pthread_cond_t term_cond_var;
   pthread_mutex_t term_mutex;
} term_struct;
typedef term_struct* term_t;

term_t term;  // global variable
```

Σχήμα 11.10: Ομαδοποίηση των μεταβλητών τερματισμού.

```
/* Global vars */
int awakened_thread = -1;
work_remains = 1;  /* true */
. . .
omp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
omp_set_lock(&term_lock);
```

Σχήμα 11.11: Προσομοίωση του condition wait στην OpenMP.

```
int MPI_Scatterv(
      void*          sendbuf          /* in   */,
      int*           sendcounts       /* in   */,
      int*           displacements    /* in   */,
      MPI_Datatype   sendtype         /* in   */,
      void*          recvbuf          /* out  */,
      int            recvcount        /* in   */,
      MPI_Datatype   recvtype         /* in   */,
      int            root             /* in   */,
      MPI_Comm       comm             /* in   */)
```

Σχήμα 11.12: Αποστολή διαφορετικού πλήθους αντικειμένων σε κάθε διαδικασία σε ένα communicator.

```
int MPI_Gatherv(
      void*          sendbuf          /* in   */,
      int            sendcount        /* in   */,
      MPI_Datatype   sendtype         /* in   */,
      void*          recvbuf          /* out  */,
      int*           recvcounts       /* in   */,
      int*           displacements    /* in   */,
      MPI_Datatype   recvtype         /* in   */,
      int            root             /* in   */,
      MPI_Comm       comm             /* in   */)
```

Σχήμα 11.13: Συγκέντρωση ενός διαφορετικού πλήθους αντικειμένων από κάθε διεργασία στον communicator.

```
int MPI_Iprobe(
           int          source       /* in  */,
           int          tag          /* in  */,
           MPI_Comm     comm         /* in  */,
           int*         msg_avail_p  /* out */,
           MPI_Status*  status_p     /* out */);
```

Σχήμα 11.14: Έλεγχος αν ένα μήνυμα είναι διαθέσιμο.

```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_request(my_stack);
   return false;  /* Still more work */
} else { /* At most 1 available tour */
   Send_rejects();  /* Tell everyone who's requested */
                    /* work that I have none        */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Messages unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
         } else if (!work_request_sent) {
            Send_work_request();  /* Request work from someone */
            work_request_sent = true;
         } else {
            Check_for_work(&work_request_sent, &work_avail);
            if (work_avail) {
               Receive_work(my_stack);
               return false;
            }
         }
      }  /* while */
   } /* Empty stack */
} /* At most 1 available tour */
```

Σχήμα 11.15: Η συνάρτηση Terminated για τη δυναμική παραλληλοποίηση της αναζήτησης δέντρου με MPI.

```
struct {
   int cost;
   int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
loc_data.rank = my_rank;

MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC, comm);
if (global_data.rank == 0) return;  /* 0 already has the best tour */
if (my_rank == 0)
   Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
   Send best tour to process 0;
```

Σχήμα 11.16: Εκτύπωση του καλύτερου γύρου.

```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_request(my_stack);
   return false;  /* Still more work */
} else { /* At most 1 available tour */
   Send_rejects();  /* Tell everyone who's requested */
                    /* work that I have none         */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Messages unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
         } else if (!work_request_sent) {
            Send_work_request();  /* Request work from someone */
            work_request_sent = true;
         } else {
            Check_for_work(&work_request_sent, &work_avail);
            if (work_avail) {
               Receive_work(my_stack);
               return false;
            }
         }
      }  /* while */
   } /* Empty stack */
} /* At most 1 available tour */
```

Σχήμα 11.17: Η συνάρτηση Terminated για τη δυναμική παραλληλοποίηση της αναζήτησης δέντρου.

```
int MPI_Pack(
      void*          data_to_be_packed    /* in      */,
      int            to_be_packed_count   /* in      */,
      MPI_Datatype   datatype             /* in      */,
      void*          contig_buf           /* out     */,
      int            contig_buf_size      /* in      */,
      int*           position_p           /* in/out  */,
      MPI_Comm       comm                 /* in      */)
```

Σχήμα 11.18: Πακετάρισμα δεδομένων σε βυφφερ συνεχόμενης μνήμης.

```
int MPI_Unpack(
      void*          contig_buf           /* in      */,
      int            contig_buf_size      /* in      */,
      int*           position_p           /* in/out  */,
      void*          unpacked_data        /* out     */,
      int            unpack_count         /* in      */,
      MPI_Datatype   datatype             /* in      */,
      MPI_Comm       comm                 /* in      */)
```

Σχήμα 11.19: Ξεπακετάρισμα δεδομένων από ένα βυφφερ συνεχόμενης μνήμης.

Time	Process 0	Process 1	Process 2
0	Out of Work Notify 1, 2 oow = 1	Out of Work Notify 0, 2 oow = 1	Working oow = 0
1	Send request to 1 oow = 1	Send Request to 2 oow = 1	Recv notify fr 1 oow = 1
2	oow = 1	Recv notify fr 0 oow = 2	Recv request fr 1 oow = 1
3	oow = 1	oow = 2	Send work to 1 oow = 0
4	oow = 1	Recv work fr 2 oow = 1	Recv notify fr 0 oow = 1
5	oow = 1	Notify 0 oow = 1	Working oow = 1
6	oow = 1	Recv request fr 0 oow = 1	Out of work Notify 0, 1 oow = 2
7	Recv notify fr 2 oow = 2	Send work to 0 oow = 0	Send request to 1 oow = 2
8	Recv 1st notify fr 1 oow = 3	Recv notify fr 2 oow = 1	oow = 2
9	Quit	Recv request fr 2 oow = 1	oow = 2

Σχήμα 11.20: Γεγονότα τερματισμού που οδηγούν σε λάθος.

Κεφάλαιο 12

Βιβλιογραφία

[1] Barry Wilkinson, Michael Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition), Prentice Hall, 2004.

[2] Ananth Grama , George Karypis , Vipin Kumar, Anshul Gupta, Introduction to Parallel Computing (2nd Edition), Addison Wesley, 2003.

[3] Peter Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann, 2011.

[4] Maurice Herlihy, Nir Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2008.

[5] Michael Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Science/Engineering/Math, 2003.

[6] Thomas Rauber, Gudula Runger, Parallel Programming: for Multicore and Cluster Systems, Springer, 2010.

[7] Thomas H. Cormen , Charles E. Leiserson, Ronald L. Rivest , Clifford Stein, Introduction to Algorithms (third edition), The MIT Press, 2009.