Βασίλειος Β. Δημακόπουλος

Παράλληλα Συστήματα και Προγραμματισμός

1η αναθεωρημένη έκδοση

Ελληνικά Ακαδημαϊκά Ηλεκτρονικά Συγγράμματα και Βοηθήματα
www.kallipos.gr

HEALLINK
Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Η εικόνα στο εξώφυλλο είναι ένα word cloud των 50 συχνότερων λέξεων από τον οδηγό του Blaise Barney, “Introduction to Parallel Computing”, ο οποίος είναι διαθέσιμος στην ιστοσελίδα `https://computing.llnl.gov/tutorials/parallel_comp/`

ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ ΚΑΙ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Βασίλειος Β. Δημακόπουλος
Αναπληρωτής Καθηγητής Πανεπιστημίου Ιωαννίνων

Παράλληλα Συστήματα και Προγραμματισμός

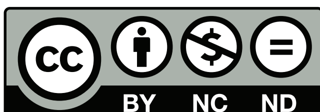
Συγγραφή
Βασίλειος Δημακόπουλος

Κριτικός αναγνώστης
Παναγιώτης Χατζηδούκας

ISBN: 978-960-603-369-8

1η αναθεωρημένη έκδοση, 2017

Copyright © ΣΕΑΒ, 2015



Το παρόν έργο αδειοδοτείται υπό τους όρους της άδειας Creative Commons Αναφορά Δημιουργού - Μη Εμπορική Χρήση - Όχι Παράγωγα Έργα 3.0. Για να δείτε ένα αντίγραφο της άδειας αυτής επισκεφτείτε τον ιστότοπο https://creativecommons.org/licenses/by-nc-nd/3.0/gr/.

ΣΥΝΔΕΣΜΟΣ ΕΛΛΗΝΙΚΩΝ ΑΚΑΔΗΜΑΪΚΩΝ ΒΙΒΛΙΟΘΗΚΩΝ
Εθνικό Μετσόβειο Πολυτεχνείο
Ηρώων Πολυτεχνείου 9, 15780 Ζωγράφου

http://www.kallipos.gr

Στη σύντροφο της ζωής μου, Δήμητρα,
με την οποία περπατάμε παράλληλα, κοντά τριάντα χρόνια τώρα.

Περιεχόμενα

Σχήματα **ix**

Προγράμματα **xiii**

Ευχαριστίες **xv**

Συντομεύσεις - Ακρωνύμια **1**

1 Εισαγωγή **3**

1.1 Παράλληλοι υπολογιστές 4
1.2 Πού υπάρχει παραλληλισμός; 10
1.2.1 Ένα παράδειγμα: το πρόβλημα των N σωμάτων 11
1.3 Παράλληλες αρχιτεκτονικές 14
1.4 Προγραμματισμός των παράλληλων υπολογιστών 17
1.4.1 Τα κύρια μοντέλα παράλληλου προγραμματισμού 18
1.5 Βασική μεθοδολογία παραλληλοποίησης 20
1.5.1 Διάσπαση σε εργασίες 22
1.6 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις 24

2 Οργάνωση Κοινόχρηστης Μνήμης **27**

2.1 Βασική οργάνωση 29
2.2 Δίκτυα διακοπτών 30
2.2.1 Διασταυρωτικός διακόπτης crossbar 31
2.2.2 Δίκτυα πολλαπλών επιπέδων—το δίκτυο Δέλτα 32
2.2.3 Άλλα δίκτυα 37
2.3 Ανάλυση επιδόσεων διασυνδέσεων 38

2.3.1 Επιδόσεις διαύλων 40
2.3.2 Επιδόσεις διασταυρωτικών διακοπτών 41
2.3.3 Επιδόσεις πολυεπίπεδων δικτύων 43
2.4 Κρυφές μνήμες και το πρόβλημα της συνοχής 44
2.5 Πρωτόκολλα πρακολούθησης 47
2.6 Πρωτόκολλα καταλόγων 52
2.7 Συνέπεια μνήμης 58
2.7.1 Ακολουθιακή συνέπεια 60
2.7.2 Χαλαρώνοντας τη συνέπεια 62
2.7.3 Συνοψίζοντας τη συνέπεια μνήμης 66
2.8 Επεξεργαστές πολλαπλών πυρήνων 66
2.9 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις 70

3 Οργάνωση Κατανεμημένης Μνήμης 75
3.1 Βασική οργάνωση 76
3.2 Η τοπολογία του δικτύου 79
3.2.1 Βασικές τοπολογίες 81
3.2.2 Σύνθετες τοπολογίες 82
3.3 Αντιστοιχίσεις τοπολογιών 86
3.3.1 Αντιστοίχιση γραμμικών γράφων και δακτυλίων στον υπερκύβο 87
3.3.2 Αντιστοίχιση πλεγμάτων και tori 90
3.3.3 Αντιστοίχιση πλήρων δυαδικών δέντρων 92
3.4 Διαδρόμηση (Routing) 94
3.4.1 Παράδειγμα: διαδρόμηση σε **d**-διάστατο υπερκύβο 95
3.5 Μεταγωγή (Switching) 96
3.5.1 Χρονική ανάλυση μεταγωγής 99
3.5.2 Σύγκριση τεχνικών μεταγωγής 101
3.6 Ομαδοποιημένοι πολυεπεξεργαστές και υπολογιστικές συστάδες 104
3.6.1 Υπολογιστικές συστάδες 105
3.7 Κατανεμημένη κοινή μνήμη 107
3.8 Πολυπύρηνοι επεξεργαστές και κατανεμημένη μνήμη 110
3.9 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις 113

4 Προγραμματισμός με Κοινόχρηστο Χώρο Διευθύνσεων 117
4.1 Γενικά περί παράλληλων προγραμμάτων 119
4.2 Βασικές δομές του μοντέλου 120
4.3 Νήματα και κοινόχρηστες μεταβλητές 124
4.4 Αμοιβαίος αποκλεισμός με κλειδαριές 128
4.5 Συγχρονισμός 131
4.5.1 Μεταβλητές συνθήκης 132

4.5.2 Κλήσεις φραγής (barriers) . . . 135
4.6 Αρχικές προγραμματιστικές τεχνικές . . . 137
4.6.1 Υπολογισμός της σταθεράς $\pi = 3.14159...$ . . . 138
4.7 Παραλληλοποίηση πολλαπλών βρόχων . . . 145
4.7.1 Πίνακας επί διάνυσμα . . . 146
4.7.2 Πίνακας επί πίνακα . . . 148
4.8 Η τεχνική της αυτοδρομολόγησης . . . 150
4.9 Προγραμματισμός με το OpenMP . . . 154
4.9.1 Νήματα και μεταβλητές . . . 155
4.9.2 Αμοιβαίος αποκλεισμός και συγχρονισμός . . . 158
4.9.3 Διαμοιρασμός εργασίας . . . 159
4.9.4 Υπολογισμός του π με OpenMP . . . 160
4.9.5 Βρόχοι με την οδηγία `for` . . . 164
4.9.6 Άλλα χαρακτηριστικά του OpenMP . . . 166
4.10 Παραλληλοποίηση «ακανόνιστων» εφαρμογών . . . 167
4.10.1 Διάσχιση λίστας . . . 169
4.10.2 Αναδρομικοί υπολογισμοί . . . 170
4.11 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις . . . 172

5 Προγραμματισμός με Μεταβίβαση Μηνυμάτων 179
5.1 Βασικές δομές . . . 181
5.1.1 Διεργασίες . . . 181
5.1.2 Αποστολή και λήψη μηνυμάτων . . . 183
5.2 Η λειτουργία της επικοινωνίας . . . 186
5.3 Παραδείγματα εφαρμογών και τεχνικών . . . 189
5.3.1 Υπολογισμός της σταθεράς $\pi = 3.14159...$ . . . 189
5.3.2 Πίνακας επί διάνυσμα . . . 192
5.3.3 Απολογισμός του βασικού μοντέλου . . . 196
5.4 Συλλογικές επικοινωνίες . . . 198
5.4.1 Συλλογικές επικοινωνίες στο MPI . . . 200
5.4.2 Υπολογισμός της σταθεράς $\pi = 3.14159...$ με συλλογικές επικοινωνίες . 203
5.4.3 Πίνακας επί διάνυσμα με συλλογικές επικοινωνίες . . . 204
5.4.4 Αποτίμηση των συλλογικών επικοινωνιών . . . 206
5.5 Χρονική επικάλυψη υπολογισμών και επικοινωνιών . . . 207
5.6 Άλλα χαρακτηριστικά του MPI . . . 214
5.7 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις . . . 217

6 Μετρικές και Επιδόσεις **221**
6.1 Χρόνος εκτέλεσης και επιτάχυνση . . . 222
6.1.1 Γραμμική και υπεργραμμική επιτάχυνση . . . 224
6.2 Αποδοτικότητα και κόστος . . . 225
6.3 Εξάρτηση από την αρχιτεκτονική . . . 226
6.4 Λίγοι επεξεργαστές—αύξηση κόκκου παραλληλίας . . . 229
6.5 Όρια στις επιδόσεις—Νόμος Amdahl . . . 234
6.6 Καταρρίπτοντας τα όρια στις επιδόσεις . . . 234
6.6.1 Το επιχείρημα . . . 235
6.6.2 Ο νόμος του Gustafson . . . 236
6.7 Επιπρόσθετος χρόνος παραλληλισμού . . . 237
6.8 Κλιμάκωση . . . 240
6.9 Σύνοψη . . . 243

Βιβλιογραφία **247**

Ευρετήριο **253**

Σχήματα

1.1 Το οικείο σειριακό υπολογιστικό μοντέλο. 5
1.2 Η εξέλιξη της συχνότητας λειτουργίας (σε MHz) των μικροεπεξεργαστών. . . 6
1.3 Παράλληλος υπολογιστής. 7
1.5 Αρχιτεκτονικές κατηγορίες υπολογιστών κατά Flynn. 15
1.6 Οργανώσεις πολυεπεξεργαστών. 16

2.1 Χονδρική οργάνωση συστήματος MIMD κοινής μνήμης. 29
2.2 Διασύνδεση με δίαυλο. 30
2.3 Διακόπτης crossbar. 31
2.4 Η πράξη 2-shuffle (α) στους διψήφιους και (β) στους τριψήφιους δυαδικούς αριθμούς. Στο σχήμα (γ) εμφανίζεται η αντίστροφη 2-shuffle για τριψήφιους δυαδικούς αριθμούς. 33
2.5 Η δομή του συμμετρικού δικτύου Δέλτα. 33
2.6 Οι 6 συνδέσεις εισόδων / εξόδων ενός crossbar 2×2. 34
2.7 Το δίκτυο Δέλτα $2^3 \times 2^3$. 34
2.8 Διαδρομές (α) και συγκρούσεις (β) στο δίκτυο Δέλτα $2^3 \times 2^3$. 36
2.9 Το δίκτυο Ωμέγα. 38
2.10 Δίκτυα baseline. 39
2.11 Επιδόσεις διαύλων και διασταυρωτικών διακοπτών για $N = 4, 8, 16, 32$ επεξεργαστές. 42
2.12 Πολυεπεξεργαστής κοινόχρηστης μνήμης με κρυφές μνήμες. 45
2.13 Διάγραμμα καταστάσεων για το Παράδειγμα 2.2. 48
2.14 Διάγραμμα καταστάσεων για το Παράδειγμα 2.3. 50
2.15 Οργάνωση με καταλόγους. 53
2.16 Πρωτόκολλο με πλήρεις καταλόγους. 55

2.17 Πρωτόκολλο με περιορισμένους καταλόγους. 56
2.18 Πρωτόκολλο με αλυσιδωτούς καταλόγους. 57
2.19 Δύο απλές διεργασίες. 58
2.20 Τμήμα αλγορίθμου αμοιβαίου αποκλεισμού. 61
2.21 Συνήθης κώδικας αναμονής για κάποια συνθήκη. 64
2.22 Ιεραρχία μνήμης σε ένα πολυεπεξεργαστικό σύστημα κοινόχρηστης μνήμης. . 67
2.23 Ιεραρχίες μνήμης σε επεξεργαστές πολλαπλών πυρήνων. 68
2.24 Αλυσιδωτή αναμονή για κάποια συνθήκη 73

3.1 Πολυεπεξεργαστής κατανεμημένης μνήμης. 77
3.2 Τυπική δομή διαδρομητή. 78
3.3 Βασικές τοπολογίες. 81
3.4 Καρτεσιανό γινόμενο δύο γράφων. 83
3.5 Σύνθετες τοπολογίες. 84
3.6 Υπερκύβοι. 85
3.7 Αντιστοίχιση του R_8 στον Q_3. 89
3.8 Αντιστοίχιση του R_6 στον Q_3. 89
3.9 Ένα πλέγμα 2×4. 90
3.10 Αντιστοίχιση του πλέγματος 2×4 στον Q_3. 91
3.11 Το πλήρες δυαδικό δέντρο με 4 επίπεδα. 92
3.12 Αντστοίχιση του πλήρους δυαδικού δέντρου με 4 επίπεδα στον Q_d. 93
3.13 Διαδρόμηση στον 3-κύβο από τον κόμβο 0 στον κόμβο 5. 95
3.14 Μεταγωγή SAF, CS και WS. 98
3.15 Χρονική ανάλυση μεταφοράς 1 flit μεταξύ δύο κόμβων. 99
3.16 Τέσσερις ταυτόχρονες διαδρομές προκαλούν αδιέξοδο σε 3-κύβο με μεταγωγή wormhole ή κυκλώματος: 1→6, 2→4, 4→2 και 6→0. 103
3.17 Οργάνωση ομαδοποιημένων πολυεπεξεργαστών. 104
3.18 Χονδρική δομή του Sequent Numa-Q. 105
3.19 Κόμβος σε σύστημα DSM. 108
3.20 Οργάνωση πολυπύρηνων επεξεργαστών Opteron. 111
3.21 Η οργάνωση του 16-πύρηνου επεξεργαστή Epiphany III. 113

4.9 Η συνάρτηση $f(x) = 4/(1+x^2)$ (αριστερά) και η προσέγγιση του εμβαδού της (δεξιά). 139
4.11 Χρόνοι εκτέλεσης για τον υπολογισμό του π σε έναν διπύρηνο υπολογιστή, με χρήση ενός διαφορετικού νήματος για κάθε επανάληψη του βρόχου for. . 141
4.13 Χρόνοι εκτέλεσης του υπολογισμού του π σε έναν διπύρηνο υπολογιστή, με χρήση δύο νημάτων, σύμφωνα με το Πρόγρ. 4.12. 143
4.15 Χρόνοι εκτέλεσης του υπολογισμού του π σε έναν διπύρηνο υπολογιστή, με χρήση δύο νημάτων και τοπικούς υπολογισμούς, σύμφωνα με το Πρόγρ. 4.14. 143

4.26 Διαμοίραση επαναλήψεων στο OpenMP με διαφορετικούς τύπους δρομολόγησης165

5.1 Σύγχρονη επικοινωνία: (α) ο αποστολέας έστειλε το μήνυμα πριν ο παραλήπτης κάνει κλήση λήψης και (β) ο παραλήπτης βρίσκεται ήδη σε κατάσταση λήψης όταν ο αποστολέας στέλνει το μήνυμα. 187
5.2 Ασύγχρονη επικοινωνία: ο αποστολέας στέλνει άμεσα το μήνυμα χωρίς να τον ενδιαφέρει η κατάσταση του παραλήπτη, και συνεχίζει τη λειτουργία του. 187
5.6 Μερικοί τύποι συλλογικών επικοινωνιών μεταξύ 4 διεργασιών, $\Delta_0 - \Delta_3$. . . . 200
5.9 Η διαδικασία αποστολής μηνύματος: (α) σε μία τυπική υλοποίηση του MPI σε συνήθη δίκτυα όπως το ethernet και (β) υλοποίηση υψηλών επιδόσεων σε γρήγορα δίκτυα. Τα διακεκομμένα βέλη αντιπροσωπεύουν πιθανές αντιγραφές δεδομένων. 210

6.1 Εύρεση του αθροίσματος 8 αριθμών 223
6.2 Επικοινωνίες για πρόσθεση 8 αριθμών σε τρισδιάστατο υπερκύβο 228
6.3 Πρόσθεση 8 αριθμών σε διδιάστατο υπερκύβο (4 κόμβων) 231
6.4 Νέα πρόσθεση 8 αριθμών σε διδιάστατο υπερκύβο 233
6.5 Αντιστοιχιση στον γραμμικό γράφο για το Πρόβλημα 6.5 245

Προγράμματα

1.4 Σειριακός κώδικας για το πρόβλημα των N-σωμάτων. 12

4.1 Δύο νήματα που συνεργάζονται για τον υπολογισμό του μέσου όρου ενός διανύσματος A, 10 στοιχείων. 122
4.2 Δημιουργία 9 νημάτων από το αρχικό νήμα και αναμονή για τον τερματισμό τους. 126
4.3 Οι συναρτήσεις που εκτελούν δύο νήματα τα οποία συνεργάζονται για τον υπολογισμό του μέσου όρου 10 στοιχείων. 130
4.4 Δύο νήματα που συγχρονίζονται μέσω μεταβλητής συνθήκης. 133
4.5 Ένα νήμα περιμένει μία συνθήκη που την επηρεάζουν N άλλα νήματα (αριστερά) και N νήματα περιμένουν μία συνθήκη που την επηρεάζει ένα άλλο νήμα (δεξιά). 134
4.6 Τελική, ορθή εκδοχή του κώδικα από το Πρόγρ. 4.3. 136
4.7 Απόδοση δικής μας ταυτότητας στα νήματα. 137
4.8 Πέρασμα πολλαπλών δεδομένων σε ένα νήμα. 138
4.10 1ος παράλληλος υπολογισμός του π: ένα νήμα ανά επανάληψη. (sas-pi1.c) . 140
4.12 2ος παράλληλος υπολογισμός του π: τόσα νήματα όσοι και οι πυρήνες. (sas-pi2.c) 142
4.14 3ος παράλληλος υπολογισμός του π: τοπικοί υπολογισμοί. (sas-pi3.c) 144
4.16 Παραλληλοποίηση δύο βρόχων (υποθέτουμε ότι το N διαιρεί ακριβώς το NPROC). (sas-mxv1.c) 147
4.17 Δεύτερη εκδοχή του Προγρ. 4.16, με παράλληλη αρχικοποίηση του αποτελέσματος και συγχρονισμό. (sas-mxv2.c) 149
4.18 Πολλαπλασιασμός τετραγωνικών πινάκων με διαχωρισμό σκακιέρας. (sas-mxm-checker.c) 151

4.19 Σκελετός προγράμματος με αυτοδρομολόγηση. 153
4.20 Εργασίες για τον υπολογισμό του π με αυτοδρομολόγηση. (sas-ss-pi.c) . . . 154
4.21 Παράδειγμα παράλληλης περιοχής σε OpenMP. 156
4.22 Πρώτη προσπάθεια για παραλληλοποίηση του υπολογισμού του π με χρήση OpenMP. (sas-omp-pi1.c) 161
4.23 Δεύτερη εκδοχή παραλληλοποίησης του υπολογισμού του π με χρήση OpenMP. (sas-omp-pi2.c) 162
4.24 Τρίτη εκδοχή παραλληλοποίησης του υπολογισμού του π με χρήση OpenMP. (sas-omp-pi3.c) 163
4.25 Τελική εκδοχή παραλληλοποίησης του υπολογισμού του π με χρήση OpenMP. (sas-omp-pi4.c) 163
4.27 Διάσχιση λίστας με εργασίες του OpenMP. 170
4.28 Αναδρομικός υπολογισμός αριθμών Fibonacci με εργασίες του OpenMP, πρώτη έκδοση. (sas-omp-fib1.c) 171
4.29 Αναδρομικός υπολογισμός αριθμών Fibonacci με εργασίες του OpenMP, δεύτερη έκδοση. (sas-omp-fib2.c) 172

5.3 Απλός υπολογισμός της σταθεράς π με MPI. (mp-pi1.c) 191
5.4 Αρχική έκδοση υπολογισμού πίνακα επί διάνυσμα με χρήση MPI. (mp-mxv1.c) 194
5.5 Δεύτερη εκδοχή του υπολογισμού πίνακα επί διάνυσμα σε MPI, με βελτιστοποίηση στις επικοινωνίες. (mp-mxv2.c) 197
5.7 Βελτιωμένος υπολογισμός της σταθεράς π. (mp-pi2-collective.c) 205
5.8 Τρίτη εκδοχή του υπολογισμού πίνακα επί διάνυσμα σε MPI, με χρήση συλλογικών επικοινωνιών. (mp-mxv3-collective.c) 206
5.10 Παράδειγμα χρήσης μη εμποδιστικών επικοινωνιών (το οποίο, όμως, έχει ένα μικρό πρόβλημα). (mp-nonblock.c) 212

Ευχαριστίες

Ξεκινώντας τις ευχαριστίες, θα ήθελα να κάνω την αρχή από τον Παναγιώτη Χατζηδούκα, συνάδελφο, συν-ερευνητή και φίλο, ο οποίος ανέλαβε την κριτική ανάγνωση του χειρογράφου. Χωρίς τη βοήθειά του, η τεχνική ποιότητα του βιβλίου θα ήταν πολύ κατώτερη. Στο φιλολογικό κομμάτι, η βοήθεια της Μαργαρίτας Λαζοκίτσιου υπήρξε πολύτιμη.

Το βιβλίο προήλθε, σε μεγάλο βαθμό, από σημειώσεις προπτυχιακών και μεταπτυχιακών μαθημάτων που σχετίζονται με τα παράλληλα συστήματα και τον προγραμματισμό τους, τα οποία διδάσκω στο Τμήμα Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων, από το 1998. Θα ήθελα να ευχαριστήσω όλους τους φοιτητές που κατά καιρούς παρακολούθησαν αυτά τα μαθήματα. Μέσα από τα σχόλιά τους και τις παρατηρήσεις τους, ελπίζω οι σημειώσεις αυτές να μετασχηματίστηκαν σε ένα αξιόλογο σύγγραμμα για τους μελλοντικούς φοιτητές.

Ένα ακόμα μεγαλύτερο ευχαριστώ, οφείλω στους δικούς μου φοιτητές, τις εργασίες των οποίων επέβλεψα όλα αυτά τα χρόνια. Μέσα από πτυχιακές, διπλωματικές, μεταπτυχιακές εργασίες και διδακτορικά, έχουν ανεβάσει την ερευνητική ομάδα Παράλληλης Επεξεργασίας (`http://paragroup.cse.uoi.gr`) στο σημείο που βρίσκεται σήμερα. Θεωρώ τον εαυτό μου τυχερό που εργάστηκαν μαζί μου. Ιδιαίτερες ευχαριστίες πρέπει να δώσω στον υποψήφιο διδάκτορα Σπύρο Αγάθο, ο οποίος όχι μόνο υπήρξε η ψυχή της ομάδας τα τελευταία χρόνια, αλλά προσφέρθηκε και ως δεύτερος κριτικός αναγνώστης για αρκετά κεφάλαια του βιβλίου.

Το τελευταίο και πιο σημαντικό ευχαριστώ θέλω να το δώσω στην σύζυγό μου και τις δύο κόρες μου, που στάθηκαν δίπλα μου υπομονετικά στα συνεχή ξενύχτια μου. Μαζί με αυτό, τους οφείλω κι ένα μεγάλο συγνώμη για όλες τις όμορφες μέρες που πέρασαν και δεν είχα τον χρόνο να πάμε μαζί μια βόλτα.

Συντομεύσεις - Ακρωνύμια

ΔΔ	Διασταυρωτικός διακόπτης
ccNUMA	Cache-Coherent Non-Uniform Memory Access
CPU	Central Processing Unit
CS	Circuit Switching
CU	Control Unit
DSM	Distributed Shared Memory
GNU	GNU is not Unix
GPU	Graphics Processing Unit
HPC	High Performance Computing
MESI	Modified, Exclusive, Shared, Invalid (πρωτόκολλο συνοχής)
MIMD	Multiple Instruction, Multiple Data
MPI	Message Passing Interface
MPP	Massively Parallel Processor
MPSoC	Multiprocessor System on Chip
MSI	Modified, Shared, Invalid (πρωτόκολλο συνοχής)
NoC	Network on Chip
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
PC	Personal Computer
POSIX	Portable Operating System Interface for Unix
PrC	Processor Consistency
PU	Processing Unit

PVM	Parallel Virtual Machine
RISC	Reduced Instruction Set Computer
RMW	Read-Modify-Write
SAF	Store-and-Forward
SCI	Scalable Coherent Interface
sDSM	Software Distributed Shared Memory
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SMP	Symmetric Multiprocessor
SoC	System on Chip
SPMD	Single Program, Multiple Data
SSI	Single System Image
UMA	Uniform Memory Access
VCT	Virtual Cut-Through
VLSI	Very Large Scale Integration
WS	Wormhole Switching

1 Εισαγωγή

Με το πρώτο αυτό κεφάλαιο ξεκινάμε το ταξίδι μας στον χώρο των παράλληλων υπολογιστικών συστημάτων. Θα έρθουμε σε επαφή με πολλές βασικές έννοιες που αφορούν σε όλες τις διαφορετικές πλευρές των παράλληλων υπολογιστών: την ανάγκη ύπαρξής τους, τις εφαρμογές που έχουν, την εσωτερική τους δομή και τον τρόπο προγραμματισμού τους. Θα δούμε ποιες είναι οι διαφορές των παράλληλων από τους κλασικούς σειριακούς υπολογιστές, αλλά και πόσο πιο απαιτητικός και πολύπλοκος είναι ο προγραμματισμός τους. Το κεφάλαιο αυτό αποτελεί προοίμιο για αυτά που θα ακολουθήσουν στα επόμενα κεφάλαια· εισάγει, τέλος, μεγάλο τμήμα της ορολογίας που θα χρησιμοποιήσουμε στο πλαίσιο της μελέτης μας.

1.1 Παράλληλοι υπολογιστές

Η πρόοδος στους ηλεκτρονικούς υπολογιστές, από τη στιγμή που έκαναν την εμφάνισή τους (δεκαετία του 1940) μέχρι σήμερα, προκαλεί θαυμασμό. Είναι ενδιαφέρον να συγκρίνει κανείς το τεχνολογικό επίτευγμα της δεκαετίας του '40, που άκουγε στο όνομα ENIAC, με ένα από τα οικονομικότερα συστήματα που μπορεί να βρει κανείς σήμερα· κοιτάξτε τον Πίν. 1.1. Ο φορητός υπολογιστής είναι τύπου «netbook» και κοστίζει λιγότερο από 250€. Αν συγκρίνει κανείς τις συχνότητες των ρολογιών, λαμβάνοντας υπόψη του ότι ένας σύγχρονος επεξεργαστής μπορεί να εκτελεί παραπάνω από μία εντολές ανά κύκλο ρολογιού, συμπεραίνει ότι η ταχύτητα έχει αυξηθεί πάνω από δέκα εκατομμύρια φορές. Ο ENIAC μπορούσε να κάνει 5000 ακέραιες προσθέσεις το δευτερόλεπτο, ενώ το ταπεινό netbook παράγει «Gflops»—δισεκατομμύρια πράξεις με πραγματικούς αριθμούς το δευτερόλεπτο.

Οι εντυπωσιακές αυτές βελτιώσεις έγιναν δυνατές σε μεγάλο βαθμό λόγω της τεχνολογίας, η οποία επέτρεψε την παραγωγή όλο και ταχύτερων υλικών σε όλο και μικρότερο χώρο· δεύτερον, λόγω της αρχιτεκτονικής των υπολογιστών, η οποία μετέτρεψε τις τεχνολογικές καινοτομίες σε περισσότερο αποδοτικά συστήματα· τρίτον, λόγω της προόδου στο λογισμικό, με την εύρεση βελτιωμένων αλγορίθμων και τρόπων υλοποιήσεων αυτών. Αν και δεν πρέπει να είμαστε απόλυτοι, δεν απέχει πολύ από την πραγματικότητα, αν πούμε ότι η τεχνολογία ήταν αυτή που έπαιξε τον πιο καθοριστικό ρόλο σε αυτή τη θεαματική βελτίωση της ταχύτητας. Είναι άλλωστε χαρακτηριστικό ότι οι «γενιές» των υπολογιστών[1] καθορίζονται βάσει της τεχνολογίας που χρησιμοποιούν.

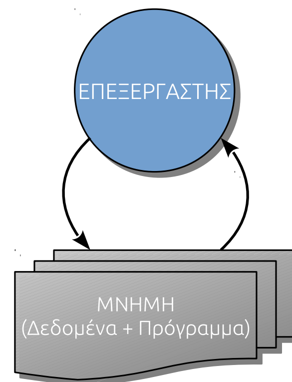
Οι υπολογιστές βασίζονται παραδοσιακά σε ένα μοντέλο προγραμματισμού / αρχιτεκτονικής το οποίο είναι γνωστό ως *μοντέλο von Neumann* ή απλώς *σειριακό*, και φαίνεται στο Σχ. 1.1. Υπάρχει μία μονάδα επεξεργασίας η οποία επικοινωνεί με το τμήμα της μνή-

[1]Η πρώτη γενιά (1940-1956) χρησιμοποιούσε λυχνίες κενού, η δεύτερη (1956-1963) τρανζίστορ, η τρίτη (1964-1971) χρησιμοποιούσε ολοκληρωμένα κυκλώματα, και η τέταρτη (1971-σήμερα) κυκλώματα υψηλής κλίμακας ολοκλήρωσης και μικροεπεξεργαστές.

	eniac (1945)	**Φορητός υπολογιστής (2015)**
Βάρος (kg)	30.000	1
Μέγεθος (m^3)	70	0.001
Κατανάλωση (Watt)	140.000	10
Μνήμη (bytes)	$\approx$ 200*	2.147.483.648
Ρολόι (MHz)	0.005	2.500

*χρησιμοποιούσε δεκαδικά ψηφία, όχι δυαδικά (bits)

Πίνακας 1.1 *Τότε και ... τώρα. Μία σύγκριση του ENIAC, του πρώτου κατά πολλούς ηλεκτρονικού υπολογιστή, με ένα οικονομικό σύστημα που μπορεί κανείς να βρει σήμερα.*

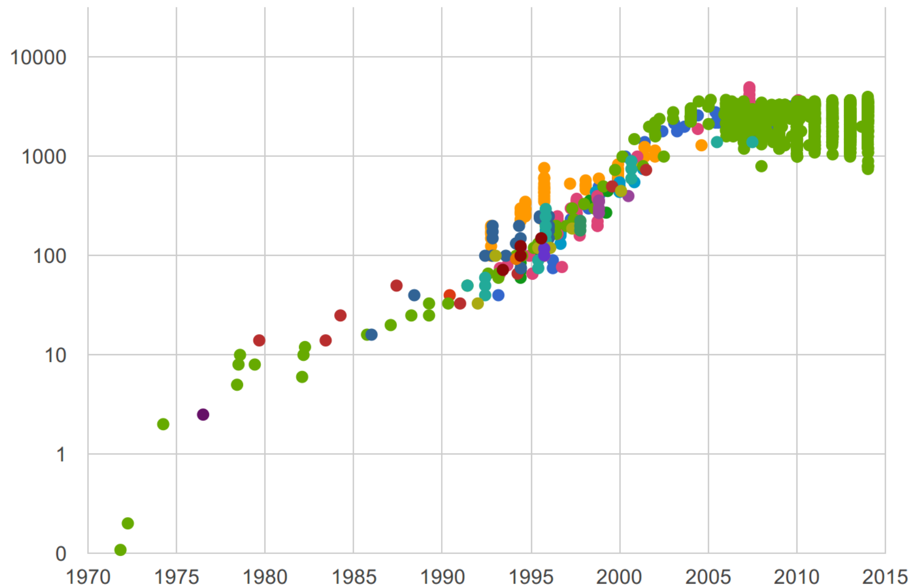


Σχήμα 1.1 *Το οικείο σειριακό υπολογιστικό μοντέλο.*

μης, εκτελώντας εντολές και τροποποιώντας δεδομένα. Το βασικό σενάριο είναι το εξής: ο επεξεργαστής λαμβάνει από τη μνήμη την επόμενη εντολή, την εκτελεί, ανάλογα με τις ενέργειες που καθορίζει, τροποποιεί κάποια δεδομένα στη μνήμη και ο κύκλος επαναλαμβάνεται. Δηλαδή, η εκτέλεση των εντολών (και επομένως οι απαιτούμενοι υπολογισμοί) γίνεται η μία μετά την άλλη και με τη σειρά που εμφανίζονται στο πρόγραμμα που εκτελείται. Με άλλα λόγια, η εκτέλεση είναι *σειριακή* ή *ακολουθιακή* και αυτό ακριβώς είναι που έχουμε συνηθίσει.

Οι βελτιώσεις στην ταχύτητα των υπολογιστών, έχουν κατά κύριο λόγο προέλθει από αντίστοιχες βελτιώσεις της τεχνολογίας και της αρχιτεκτονικής των δύο βασικών υποσυστημάτων τους: του επεξεργαστή και (λιγότερο) της μνήμης. Παρ' όλα αυτά, η βασική οργάνωση έχει παραμείνει η ίδια. Το ερώτημα που τίθεται είναι το εξής: πόσο πιο γρήγορος μπορεί να γίνει ο επεξεργαστής (δηλαδή, πόσο πιο γρήγορα μπορούν να εκτελεστούν τα προγράμματα); Αν και ίσως υπάρχουν ακόμα περιθώρια βελτίωσης, έχει από καιρό αρχίσει να διαφαίνεται ότι υπάρχει κάποιο όριο. Κατ' αρχήν, υπάρχουν φυσικοί περιορισμοί. Για παράδειγμα, η ταχύτητα με την οποία κινούνται τα ηλεκτρόνια μέσα στα ηλεκτρονικά κυκλώματα δεν μπορεί να ξεπεράσει την ταχύτητα του φωτός. Όσο και αν φαίνεται παράξενο, ο φυσικός αυτός περιορισμός έχει αρχίσει να κάνει όλο και εντονότερη την εμφάνισή του, σε σημείο που να είναι πλέον αποδεκτό ότι δεν θα αργήσει η εποχή κατά την οποία οι βελτιώσεις στις ταχύτητες δεν θα είναι αυτές που έχουμε συνηθίσει μέχρι τώρα. Και αυτό γίνεται άμεσα αντιληπτό στο Σχ. 1.2, το οποίο παρουσιάζει την εξέλιξη των συχνοτήτων λειτουργίας των επεξεργαστών (οι κουκκίδες αντιπροσωπεύουν ξεχωριστές οικογένειες εμπορικών μικροεπεξεργαστών). Αν και οι επιδόσεις δεν εξαρτώνται μόνο από τη συχνότητα του ρολογιού, εντούτοις είναι φανερό ότι μετά το 2005 υπάρχει πολύ μεγάλη στασιμότητα.

Εκτός από τα φυσικά όρια, η συνεχής βελτίωση της τεχνολογίας οδηγεί σε προβλήματα τα οποία δεν είχαμε να αντιμετωπίσουμε στο παρελθόν. Ως χαρακτηριστικό παράδειγμα, η συνεχής προσπάθεια για αύξηση των συχνοτήτων λειτουργίας και ταυτόχρονα η



(Τα δεδομένα έχουν προέλθει από τη βάση δεδομένων CPUDB, `http://cpudb.stanford.edu`)

Σχήμα 1.2 *Η εξέλιξη της συχνότητας λειτουργίας (σε MHz) των μικροεπεξεργαστών.*

σμίκρυνση των φυσικών διαστάσεων των ηλεκτρονικών κυκλωμάτων, μας έφερε μπροστά σε ένα τεχνολογικό αδιέξοδο: την εκθετική αύξηση της κατανάλωσης ενέργειας. Κάπου γύρω στο 2005, το πρόβλημα αυτό αποδείχτηκε σχεδόν αδύνατον να επιλυθεί, τουλάχιστον με τις μέχρι τότε (και μέχρι τώρα) τεχνολογικές μας δυνατότητες.

Πώς όμως θα κατορθώσουμε να λύσουμε ορισμένα πραγματικά υπολογιστικά προβλήματα τα οποία απαιτούν εξαιρετικά αυξημένη υπολογιστική ισχύ, εφόσον η τεχνολογία δεν φαίνεται να μπορεί να ανταπεξέρχεται για πάντα; Η μόνη πρακτική και ρεαλιστική λύση (διότι η άλλη λύση είναι η πλήρης αλλαγή υπολογιστικού μοντέλου, π.χ. με κβαντικούς υπολογιστές[2]) φαίνεται ότι βρίσκεται στην απομάκρυνση από το σειριακό μοντέλο: αντί να υπάρχει *ένας* επεξεργαστής ο οποίος είναι αναγκασμένος να εκτελεί μία εντολή τη φορά, γιατί να μην υπάρχουν *περισσότεροι* (N) επεξεργαστές, ώστε να εκτελούνται περισσότερες (N) εντολές ταυτόχρονα; Η ιδέα δίνεται στο Σχ. 1.3.

Με αυτόν τον τρόπο, μπορούμε (θεωρητικά τουλάχιστον) να φτιάξουμε έναν υπολογιστή όσο γρήγορο επιθυμούμε, αυξάνοντας τον αριθμό των επεξεργαστών όσο χρειάζεται. Και μάλιστα, χωρίς να χρησιμοποιούμε ακριβούς επεξεργαστές· αρκεί η χρήση φθηνών και δοκιμασμένων. Αυτή είναι και η βασική ιδέα πίσω από τους *παράλληλους* υπολογιστές:

> *μία συλλογή από επεξεργαστικές μονάδες που επικοινωνούν*
> *και συνεργάζονται για τη λύση ενός προβλήματος.*

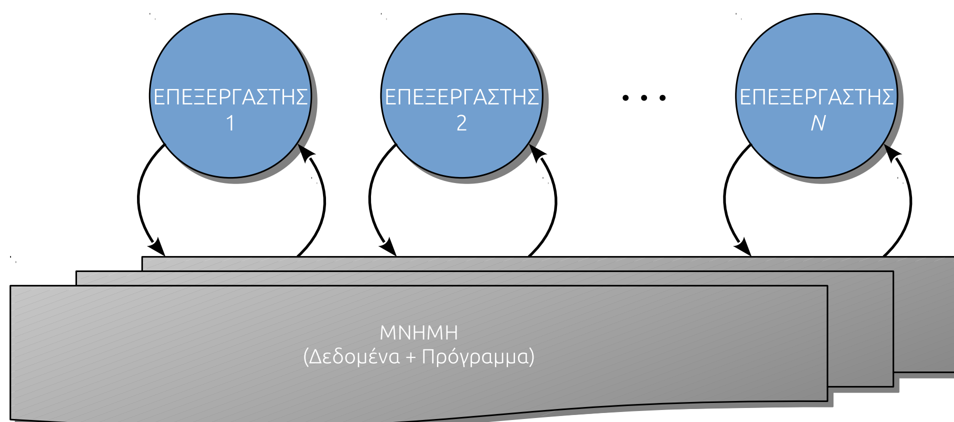
[2]Αν σας κέντρισαν την περιέργεια, μία κλασική πηγή για τους κβαντικούς υπολογιστές, πέρα από τις πληροφορίες που μπορείτε να βρείτε στο διαδίκτυο, είναι το βιβλίο των Nielsen και Chuang [NiCh11].

Η ιδέα αυτή φυσικά δεν είναι καινούργια, λέγεται μάλιστα ότι διατυπώθηκε από τον ίδιο τον von Neumann, ως μοιραία βελτίωση της αρχιτεκτονικής του υπολογιστή. Ωστόσο, σε αυτή την ιδέα βασίστηκε και η βιομηχανία για να λύσει το προαναφερθέν τεχνολογικό πρόβλημα της κατανάλωσης ενέργειας. Έτσι, κάπου στα μέσα της δεκαετίας του 2000, στην αγορά των εμπορικών επεξεργαστών συντελέσθηκε η λεγόμενη «επανάσταση» των *πολυπύρηνων* επεξεργαστών (multicore processors): δύο ή παραπάνω επεξεργαστικοί «πυρήνες» μέσα στο ίδιο ολοκληρωμένο κύκλωμα, κι ας έχει ο καθένας τους χαμηλότερες επιδόσεις από ότι ένας αντίστοιχος σειριακός («μονοπύρηνος») επεξεργαστής!

Όσο απλή και αν φαίνεται, όμως, η ιδέα των πολλαπλών επεξεργαστών ή πυρήνων (οι δύο όροι θα είναι ισοδύναμοι για μας σε αυτό το κεφάλαιο), κρύβει από πίσω της μία σειρά από ερωτήματα που δεν έχουν πάντα εύκολη απάντηση:

- Πόσα και πόσο ισχυρά πρέπει να είναι τα επεξεργαστικά στοιχεία;
- Πώς επικοινωνούν;
- Πώς συντονίζονται και πώς συνεργάζονται στην εκτέλεση μιας εργασίας;
- Πώς διαμοιράζονται οι εργασίες;
- Για ποια προβλήματα απαιτούνται τέτοιου είδους υπολογιστές;
- Πώς διαφοροποιείται πλέον ο προγραμματισμός τους;

Οι απαντήσεις στα παραπάνω ερωτήματα είναι αλληλοεξαρτώμενες. Ενώ στο σειριακό μοντέλο ο προγραμματιστής δεν γνώριζε (ούτε χρειαζόταν να γνωρίζει) τις αρχιτεκτονικές λεπτομέρειες της μνήμης και του επεξεργαστή, θα δούμε ότι το εντελώς αντίθετο συμβαίνει με τον προγραμματισμό των παράλληλων μηχανών. Ο προγραμματισμός των συστημάτων αυτών έχει αρκετές και σημαντικές διαφοροποιήσεις από τον κλασικό σειριακό προγραμματισμό που έχουμε συνηθίσει και γενικά είναι αρκετά δυσκολότερος. Είναι επο-



Σχήμα 1.3 *Παράλληλος υπολογιστής.*

μένως αναγκαίο, να δούμε ορισμένα βασικά σημεία στην αρχιτεκτονική των παράλληλων υπολογιστών, ώστε να κατανοηθούν οι βασικές αρχές του παράλληλου προγραμματισμού. Πριν όμως προχωρήσουμε, ας δούμε σύντομα μερικές απαντήσεις στα τέσσερα πρώτα από τα παραπάνω ερωτήματα. Με τα τελευταία δύο ερωτήματα θα ασχοληθούμε στις Ενότητες 1.2 και 1.4.

Επεξεργαστές — Ο αριθμός των επεξεργαστών μπορεί να είναι από δύο έως και ποσότητες που ξεπερνούν το εκατομμύριο. Είναι λογικό ότι στο άνω άκρο όπου έχουμε συστήματα με χιλιάδες, εκατοντάδες χιλιάδες και εκατομμύρια επεξεργαστές, οι επεξεργαστές που χρησιμοποιούνται είναι σχετικά απλοί και οικονομικοί, ώστε το κόστος της μηχανής να παραμένει όσο το δυνατό σε λογικά επίπεδα. Για να εκμεταλλευτεί κάποιος στο έπακρο τους υπολογιστές αυτούς, που είναι και γνωστοί ως *μαζικά παράλληλοι*, πρέπει να έχει εφαρμογές που μπορούν να διασπαστούν σε πάρα πολλά τμήματα, ώστε να εκτελεστούν ταυτόχρονα. Για τέτοιες, εξειδικευμένες εφαρμογές, οι μαζικά παράλληλοι υπολογιστές είναι ό,τι ταχύτερο μπορεί να υπάρξει. Παρόμοια είναι και τα συστήματα που αποτελούνται από μεγάλο πλήθος συστάδων με αυτόνομους κόμβους από εμπορικούς επεξεργαστές, οι οποίοι όμως, μπορούν να προγραμματιστούν σαν να είναι ένας ενιαίος υπολογιστής. Πρόσφατα παραδείγματα μεγάλων συστημάτων αποτελούν το σύστημα IBM Sequoia, το οποίο διαθέτει 1.572.864 πυρήνες (πάνω από 90.000 δεκαεξαπύρηνοι επεξεργαστές), ενώ το ισχυρότερο σύστημα για το 2015 στη λίστα «Top500» των 500 κορυφαίων υπολογιστών στον κόσμο, είναι το Tianhe-2 με 3.120.000 πυρήνες!

Στο άλλο άκρο, συναντάμε συστήματα με λίγους επεξεργαστές ή πυρήνες οι οποίοι συνήθως είναι αρκετά ισχυροί. Είναι προφανές ότι υπάρχει, σε γενικές γραμμές, κάποιος συμβιβασμός μεταξύ ισχύος και αριθμού επεξεργαστών που μπορούν να ενσωματωθούν στο σύστημα. Η πιο διαδεδομένη συνδεσμολογία των επεξεργαστών στα «μικρά» συστήματα είναι σε έναν κοινό δίαυλο, όπως θα δούμε σε επόμενο κεφάλαιο. Εξαιτίας της περιορισμένης χωρητικότητας του διαύλου, ο αριθμός των επεξεργαστών δεν μπορεί να πάρει πολύ μεγάλες διαστάσεις. Πάντως, είναι σημαντικό να αναφέρουμε ότι πλέον, αυτή η κατηγορία παράλληλων αρχιτεκτονικών είναι παρούσα σε όλους τους προσωπικούς υπολογιστές (PC). Ο λόγος είναι ότι, εδώ και αρκετά χρόνια, ακόμα και το οικονομικότερο οικιακό PC είναι τουλάχιστον διπύρηνο. Το ίδιο ισχύει πλέον και για πολύ μικρότερα υπολογιστικά συστήματα, όπως π.χ. τα έξυπνα κινητά τηλέφωνα.

Επικοινωνία — Η επικοινωνία γίνεται βασικά με δύο τρόπους: μέσω κοινόχρηστης μνήμης ή μέσω απευθείας συνδέσεων μεταξύ των επεξεργαστών. Στην πρώτη περίπτωση, όλοι οι επεξεργαστές μοιράζονται την ίδια μνήμη και, επομένως, ο ένας μπορεί να δει τι έχει γράψει εκεί ο άλλος. Στη δεύτερη περίπτωση, δεν υπάρχει κοινόχρηστη μνήμη και η επικοινωνία γίνεται μέσω μηνυμάτων που ανταλλάσσουν οι επεξεργαστές που πρέπει να επικοινωνήσουν. Τα μηνύματα μεταφέρονται μέσω ενός δικτύου διασύνδεσης των επεξεργαστών. Κάθε μία από τις μεθόδους αυτές έχει θετικά και αρνητικά στοιχεία όπως θα δούμε, αλλά το πιο

σημαντικό είναι ότι επηρεάζει άμεσα τον τρόπο που προγραμματίζεται το σύστημα.

Συντονισμός και συνεργασία — Προκειμένου να διεκπεραιωθεί η εργασία την οποία έχει αναλάβει ο παράλληλος υπολογιστής ως σύνολο, θα πρέπει να υπάρξει συντονισμός μεταξύ των «εργατών» που θα την εκτελέσουν. Δεν μπορεί, με άλλα λόγια, ο κάθε επεξεργαστής να εργάζεται εντελώς ανεξάρτητα από τον άλλο. Μια άμεση αναλογία μπορούμε να δούμε στα ομαδικά αθλήματα (π.χ. ποδόσφαιρο): κάθε παίχτης εργάζεται μόνος του και ταυτόχρονα με τους άλλους. Όμως, προκειμένου να επιτευχθεί ο σκοπός (π.χ. το γκολ), είναι απαραίτητο σε συγκεκριμένες στιγμές να εφαρμοστεί από όλους κάποια προαποφασισμένη στρατηγική και οι κατάλληλοι παίχτες να βρίσκονται στις κατάλληλες θέσεις. Κάτι εντελώς ανάλογο συμβαίνει και με τη λύση ενός προβλήματος όπου συμμετέχουν πολλοί επεξεργαστές.

Απαιτείται επομένως, συντονισμός και συνεργασία. Το πώς επιτυγχάνεται αυτό εξαρτάται από το σύστημα επικοινωνίας που υπάρχει. Θα δούμε πώς διαφοροποιούνται τα πράγματα μεταξύ των υπολογιστών κοινόχρηστης μνήμης και αυτών που διαθέτουν άμεσες συνδέσεις. Βλέπουμε λοιπόν για πρώτη φορά ότι η απάντηση σε μία ερώτηση δεν είναι μεμονωμένη αλλά εξαρτάται από την απάντηση σε κάποια άλλη από τις ερωτήσεις που θέσαμε παραπάνω.

Διαμοίραση εργασιών — Αυτό είναι ίσως το δυσκολότερο από τα ερωτήματα που θέσαμε, καθώς έχει πολλές πτυχές. Η απάντηση συνήθως δεν είναι μοναδική, αφού εξαρτάται και αυτή από τις απαντήσεις στα άλλα πέντε ερωτήματα. Εξαρτάται από το πρόβλημα, από τον αριθμό των επεξεργαστών, τη μέθοδο επικοινωνίας και τις χρονικές καθυστερήσεις που αυτή συνεπάγεται, καθώς και από το προγραμματιστικό μοντέλο που χρησιμοποιείται.

Αν υποθέσουμε ότι διαθέτουμε πολλούς επεξεργαστές, είναι λογικό να σκεφτούμε ότι το πρόβλημα πρέπει να χωριστεί σε όσο το δυνατόν περισσότερα τμήματα τα οποία θα λυθούν ταυτόχρονα. Στην περίπτωση αυτή, μιλάμε για *λεπτό κόκκο παραλληλίας* (fine-grained parallelism) με την έννοια ότι ζητάμε καθένας από τους πολλούς επεξεργαστές να εκτελέσει όσο το δυνατόν λιγότερες εντολές προκειμένου να τελειώσει ταχύτατα η λύση του προβλήματος. Για παράδειγμα, αν ένα πρόγραμμα έχει N εντολές και διαθέτουμε N επεξεργαστές, το ιδεώδες θα ήταν κάθε επεξεργαστής να εκτελέσει μία εντολή ταυτόχρονα με τους άλλους, ώστε να ολοκληρωθεί η εκτέλεση του προγράμματος σε ένα μόνο βήμα.

Ξεχνάμε, όμως, ότι στις περισσότερες περιπτώσεις απαιτείται και συνεργασία για τη λύση του προβλήματος, κάτι που συνεπάγεται επικοινωνία μεταξύ των επεξεργαστών. Η επικοινωνία, όμως, κοστίζει ακριβά σε χρόνο, σε σημείο μάλιστα που προσπαθούμε όσο το δυνατό να την αποφύγουμε. Αυτό μπορεί να γίνει με το να συγκεντρώσουμε περισσότερους υπολογισμούς σε κάθε επεξεργαστή, ώστε η ανάγκη για επικοινωνία του με τους υπόλοιπους να εμφανίζεται σπάνια. Φτάνουμε έτσι στο άλλο άκρο, αυτό του *χοντρού κόκκου παραλληλίας* (coarse-grained parallelism), όπου ο κάθε επεξεργαστής εκτελεί μεγάλο αριθμό εντολών, ακόμα και αν αυτό σημαίνει ότι δεν θα εκμεταλλευτούμε όλους τους επεξεργαστές που υπάρχουν.

Η σωστή τακτική στη διαμοίραση εργασιών σχεδόν πάντα βρίσκεται κάπου στο ενδιάμεσο. Όπως έχουμε ξαναπεί, τόσο το ίδιο το πρόβλημα όσο και τα χαρακτηριστικά της αρχιτεκτονικής του συστήματος είναι αυτά που καθορίζουν την βέλτιστη λύση.

1.2 Πού υπάρχει παραλληλισμός;

Θα δούμε τώρα σύντομες απαντήσεις στα δύο τελευταία ερωτήματα που τέθηκαν στην Ενότητα 1.1. Συγκεκριμένα, θα αναφερθούμε σε μερικές εφαρμογές που απαιτούν την ισχύ των παράλληλων υπολογιστών, αλλά κυρίως θα δούμε, μέσω ενός παραδείγματος, πώς μπορούμε να σκεφτούμε προκειμένου να προγραμματίσουμε παράλληλα.

Το πρώτο πράγμα που μας έρχεται στο νου όταν διατυπώνουμε τη βασική ιδέα του παραλληλισμού, δηλαδή τη δυνατότητα εκτέλεσης ταυτόχρονα πολλών υπολογισμών, είναι το εξής ερώτημα: πώς είναι δυνατόν να γίνει αυτή η ταυτόχρονη εκτέλεση από τη στιγμή που όλοι οι τρόποι που γνωρίζουμε για τη λύση προβλημάτων είναι σειριακοί; Κι όμως, τα πράγματα δεν είναι τόσο δύσκολα. Ας πάρουμε ως παράδειγμα το παρακάτω πρόγραμμα το οποίο προσθέτει δύο διανύσματα k στοιχείων το καθένα:

```
for (i = 0; i < k; i = i+1)
  c[i] = a[i]+b[i];
```

Δεν είναι δύσκολο να δούμε ότι οι υπολογισμοί που γίνονται είναι *ανεξάρτητοι* μεταξύ τους: ο υπολογισμός `c[1] = a[1]+b[1]` χρησιμοποιεί εντελώς διαφορετικά στοιχεία των διανυσμάτων απ' ότι οι `c[2] = a[2]+b[2]`, `c[3] = a[3]+b[3]` κλπ. Είναι επομένως φανερό ότι ο υπολογισμός του `c[1]` δεν εμπλέκεται καθόλου στον υπολογισμό του `c[2]` ούτε του `c[3]` κλπ. και ως επακόλουθο, οι υπολογισμοί αυτοί μπορούν να γίνουν *με οποιαδήποτε σειρά*, ή ακόμα καλύτερα, *ταυτόχρονα* («παράλληλα»). Αν είχαμε k επεξεργαστές, ο καθένας θα μπορούσε να υπολογίσει διαφορετικό στοιχείο του διανύσματος `c[]` και με ένα μόνο βήμα θα είχαμε υπολογίσει όλα τα ζητούμενα στοιχεία, τη στιγμή που η σειριακή έκδοση απαιτεί k βήματα.

Τα πράγματα βέβαια, δεν είναι πάντα τόσο απλά. Σίγουρα, υπάρχουν εφαρμογές στις οποίες, όπως παραπάνω, υπάρχει ο λεγόμενος «*εύκολος*» ή «*προφανής*» παραλληλισμός. Ένα παράδειγμα προφανούς παραλληλισμού έχουμε στους υπολογισμούς τύπου Monte Carlo που χρησιμοποιούνται ευρέως σε αρκετά επιστημονικά προβλήματα: εδώ, κάποιος πολύπλοκος υπολογισμός επαναλαμβάνεται πολλές φορές με διαφορετικές τυχαίες παραμέτρους κάθε φορά, και από τα αποτελέσματα λαμβάνεται ο μέσος όρος. Κάθε μία από τις επαναλήψεις του υπολογισμού είναι ανεξάρτητη από τις άλλες και επομένως, έχουμε μια προφανή μέθοδο παράλληλης εκτέλεσής τους.

Άλλες εφαρμογές που απαιτούν τεράστια υπολογιστική ισχύ, η οποία δεν μπορεί να καλυφθεί ακόμα και από τον ταχύτερο σειριακό υπολογιστή, είναι κάποια επιστημονικά

κυρίως προβλήματα τα οποία και έχουν χαρακτηριστεί ως «μεγάλες προκλήσεις» (grand challenges) και περιλαμβάνουν:

- τη συμπεριφορά σωματιδίων (π.χ. δυναμική των μορίων),
- τη μελέτη των κυμάτων των ωκεανών και την πρόβλεψη καιρού,
- τα σεισμικά μοντέλα,
- το σχεδιασμό κυκλωμάτων VLSI με τη βοήθεια του υπολογιστή,
- την εξέλιξη των γαλαξιών.

Από αυτές τις εφαρμογές, άλλες διαθέτουν στη λύση τους μεγάλο βαθμό παραλληλίας και άλλες λιγότερο.

1.2.1 Ένα παράδειγμα: το πρόβλημα των N σωμάτων

Ας δούμε όμως, ένα μικρό παράδειγμα από τις μεθόδους που χρησιμοποιούνται για τη λύση αυτών των προβλημάτων. Η δυναμική των μορίων, η προσομοίωση των ρευμάτων στους ωκεανούς κλπ., βασίζονται στη μελέτη ενός μεγάλου αριθμού σωμάτων τα οποία αλληλοεπηρεάζονται και μεταβάλουν την κατάσταση του συστήματος την οποία θέλουμε να γνωρίζουμε κάθε στιγμή. Όταν λέμε ότι αλληλοεπηρεάζονται, εννοούμε ότι μεταξύ τους υπάρχουν ηλεκτρικές, μαγνητικές ή βαρυτικές δυνάμεις, ανάλογα με το πρόβλημα. Στην καρδιά των προβλημάτων αυτών υπάρχει, λοιπόν, μια συλλογή από σώματα (έστω N), καθώς και δυνάμεις f_{ij} που ασκούνται μεταξύ των σωμάτων i και j (για κάθε ζεύγος σωμάτων). Η κατάσταση αυτή είναι γνωστή ως *πρόβλημα των N σωμάτων* (N-body problem). Οι δυνάμεις που ασκούνται είναι ικανές να μετατοπίσουν τα σώματα σε νέες θέσεις και να μεταβάλουν έτσι το υπό μελέτη σύστημα.

Για να γίνουμε πιο συγκεκριμένοι, ας θεωρήσουμε το πρόβλημα της εξέλιξης των γαλαξιών κάτω από την επήρεια των βαρυτικών δυνάμεων. Τα σώματα στην περίπτωση αυτή, είναι αστέρες, πλανήτες κλπ. Η δύναμη που ασκείται μεταξύ δύο ουρανίων σωμάτων δίνεται από το νόμο του Νεύτωνα για την παγκόσμια έλξη:

$$\mathbf{f}_{ij} = \frac{G m_i m_j (\mathbf{x}_i - \mathbf{x}_j)}{|\mathbf{x}_i - \mathbf{x}_j|^3} \tag{1.1}$$

όπου $\mathbf{x}_k$ είναι η θέση (διάνυσμα στον τριδιάστατο χώρο) και m_k η μάζα του k-οστού σώματος. Η συνολική δύναμη που ασκείται στο i-οστό σώμα είναι το (διανυσματικό) άθροισμα αυτών των δυνάμεων,

$$\mathbf{F}_i = \sum_{j=0}^{N-1} \mathbf{f}_{ij}. \tag{1.2}$$

```
/* [0 .. MAX_TIME]: the time interval we want to study
 * f(i,j):          force between i and j (using 1.1)
 * xnew(k,F[k]):    new position of k (using 1.3)
 *
 * In the following loop, time advances by deltat.
 */
for (t = 0; t < MAX_TIME; t = t + deltat) {
  for (i = 0; i < N; i++) {
    F[i] = zero();      /* calculate sum (using 1.2) */
    for (j = 0; j < N; j++)
      F[i] = add( F[i], f(i,j) );
  }
  for (k = 0; k < N; k++)
    xnew( k, F[k] );    /* calculate new positions */
}
```

Πρόγραμμα 1.4 *Σειριακός κώδικας για το πρόβλημα των N-σωμάτων.*

Μετά την πάροδο χρόνου dt, η νέα θέση του σώματος υπολογίζεται από τη στιγμιαία επιτάχυνση, με βάση το δεύτερο νόμο του Νεύτωνα για την κίνηση:

$$\mathbf{F}_i = m_i \frac{d^2 \mathbf{x}_i}{dt^2}. \tag{1.3}$$

Στη συνέχεια, οι νέες τιμές των $\mathbf{x}_i$ χρησιμοποιούνται για τον υπολογισμό των νέων $\mathbf{f}_{ij}$ και ο κύκλος επαναλαμβάνεται για όσα χρονικά βήματα είναι απαραίτητο.

Με βάση τα παραπάνω, ένας απλός (αλλά όχι και βέλτιστος) σειριακός αλγόριθμος για τον υπολογισμό των νέων θέσεων των σωμάτων μετά την πάροδο κάποιου χρόνου, δίνεται στο Πρόγραμμα 1.4. Προσέξτε κάτι που δεν είναι άμεσα ορατό: για μία ακριβή μελέτη του δικού μας γαλαξία το N (ο αριθμός των σωμάτων) είναι της τάξης των εκατοντάδων δισεκατομμυρίων. Επίσης, η ακρίβεια των υπολογισμών είναι άμεσα συνδεδεμένη με το μέγεθος του χρονικού βήματος Δt (σταθερά `deltat` στο πρόγραμμα). Όσο μικρότερο είναι, τόσο μεγαλύτερη και η ακρίβεια των υπολογισμών, αλλά ταυτόχρονα, τόσο περισσότεροι υπολογισμοί απαιτούνται για τη μελέτη των ίδιων χρονικών διαστημάτων. Π.χ. αν για ένα χρονικό βήμα (μία επανάληψη του βρόχου `t`) απαιτείται 1 λεπτό υπολογισμών (που είναι εξωπραγματικά μικρό αν αναλογιστούμε ότι πρέπει να εξετάσουμε δισεκατομμύρια σώματα), και το βήμα Δt το έχουμε καθορίσει να είναι 24 ώρες, για να μελετήσουμε (με όχι και τόσο μεγάλη ακρίβεια, προφανώς) έναν αιώνα εξέλιξης στο γαλαξία, θα χρειαστεί κάτι λιγότερο από ένας μήνας υπολογισμών! Μιλάμε επομένως, για προβλήματα με τεράστιες απαιτήσεις τόσο σε μνήμη όσο και σε ταχύτητα.

Το Πρόγραμμα 1.4 παραλληλοποιείται εύκολα αν προσέξουμε ότι οι επαναλήψεις του βρόχου `i` είναι ανεξάρτητες μεταξύ τους. Μπορούμε επομένως, να έχουμε έναν επεξεργαστή να υπολογίζει ένα ξεχωριστό `F[i]`, ανεξάρτητα από τους άλλους, απαιτώντας

έτσι χρόνο N φορές λιγότερο από ότι το σειριακό πρόγραμμα. Αν το N, όμως, είναι όπως είπαμε, υπερβολικά μεγάλο και δεν διαθέτουμε N, παρά μόνο $P < N$ επεξεργαστές, τότε ο καθένας μπορεί να αναλάβει N/P σώματα και να έχουμε μια βελτίωση κατά P φορές στον απαιτούμενο χρόνο.

Μια άλλη λεπτομέρεια που παρακάμψαμε, είναι το γεγονός ότι η συνάρτηση `f()` χρησιμοποιεί τις τιμές των $\mathbf{x}_j$, κάποιες από τις οποίες όμως, υπολογίζονται από άλλους επεξεργαστές. Επομένως, γίνεται φανερή η ανάγκη επικοινωνίας των επεξεργαστών σε κάποιο σημείο, ώστε σε όλους να «γνωστοποιηθούν» οι τιμές αυτές, κάτι που δεν έχουμε να σκεφτούμε στο σειριακό πρόγραμμα. Για παράδειγμα, επειδή ο βρόχος `k` υπολογίζει νέες θέσεις $\mathbf{x}_j$, θα πρέπει να σιγουρευτούμε ότι κανένας επεξεργαστής δεν θα προχωρήσει στην επόμενη χρονική στιγμή `t`, αν δεν έχει ολοκληρωθεί ο βρόχος `t` ακόμα και για το τελευταίο σώμα.

Ο παραπάνω αλγόριθμος, όμως, δεν είναι ο καλύτερος. Εκτελεί διπλάσια εργασία απ' ότι είναι αναγκαία και αυτό γιατί οι δυνάμεις $\mathbf{f}_{ij}$ είναι συμμετρικές ($\mathbf{f}_{ij} = -\mathbf{f}_{ji}$). Ο βρόχος του `j` μπορεί να αντικατασταθεί από τον καλύτερο:

```
for (j = 0; j < i; j++) {
  g = f(i,j);
  F[i] = add( F[i], g );
  F[j] = subtract( F[j], g );
}
```

Όμως, η αλλαγή αυτή ανατρέπει πολλά από τη μέθοδο παραλληλοποίησης που περιγράψαμε παραπάνω:

(α) Διαφορετικό `i` παράγει διαφορετικό αριθμό επαναλήψεων στο βρόχο `j`. Αυτό έχει ως αποτέλεσμα ότι αν μοιράσουμε την εργασία, όπως πριν, σε N επεξεργαστές, *δεν πρόκειται να εργαστούν όλοι για το ίδιο διάστημα*. Ο επεξεργαστής που αναλαμβάνει το `i=1` έχει μόνο μία επανάληψη στο βρόχο `j`, ενώ ο επεξεργαστής που έχει αναλάβει το σώμα `i=N-1` θα έχει $N-1$ επαναλήψεις στον ίδιο βρόχο. Είναι επομένως βέβαιο, ότι πολλοί επεξεργαστές θα μένουν άπραγοι, ενώ άλλοι θα εργάζονται και πλέον δεν υπάρχει περίπτωση να τελειώσουμε στο $1/N$ του χρόνου, όπως υπολογίζαμε.

(β) Οι επαναλήψεις του βρόχου `i` δεν είναι πλέον ανεξάρτητες μεταξύ τους, αφού ένα συγκεκριμένο στοιχείο του `F[]` επηρεάζεται από παραπάνω από μία επαναλήψεις του βρόχου `i`.

Φτάσαμε λοιπόν σε ένα σημείο όπου η παραλληλοποίηση του κώδικα δεν είναι τόσο απλή υπόθεση. Δεν πρόκειται να προχωρήσουμε παραπέρα και να δείξουμε πώς υλοποιείται η παράλληλη έκδοση του αλγόριθμου. Θα πούμε μόνο ότι στην ουσία, για να μεταφέρουμε τον παραπάνω κώδικα σε έναν παράλληλο υπολογιστή, θα πρέπει να ξεφύγουμε εντελώς από τη σειριακή λύση και να σκεφτούμε ίσως και έναν νέο τρόπο επίλυσης του προβλήματος.

Ως συμπέρασμα, θα πρέπει να ειπωθεί ότι υπάρχουν πολλές εφαρμογές υψηλού επιστημονικού, κοινωνικού και ανθρωπιστικού ενδιαφέροντος, οι οποίες απαιτούν τεράστια υπολογιστική ισχύ και μονόδρομος για τη λύση τους είναι ο παραλληλισμός. Ο παραλληλισμός αυτός άλλοτε είναι προφανής και άλλοτε οδηγεί σε μεθοδολογίες που, ίσως, διαφέρουν πολύ από τις σειριακές. Θα δούμε μάλιστα, ότι ιδιαίτερα σε αυτές τις περιπτώσεις, βασική παράμετρος στη διαδικασία της παραλληλοποίησης είναι η αρχιτεκτονική δομή του συστήματος.

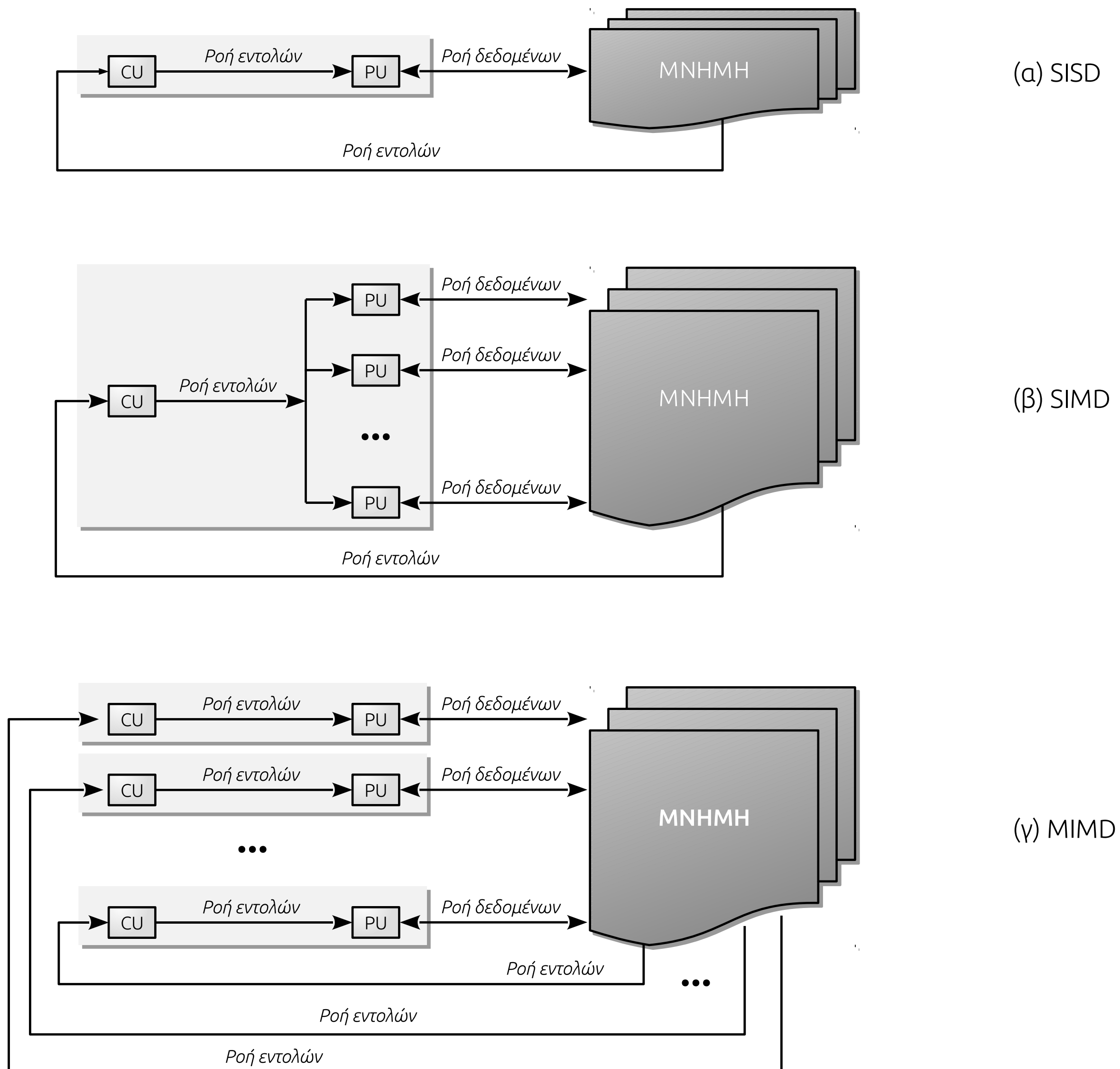
1.3 Παράλληλες αρχιτεκτονικές

Στην ενότητα αυτή, θα κάνουμε μία σύντομη εισαγωγή στις αρχιτεκτονικές των παράλληλων υπολογιστών. Όπως έχουμε πει αρκετές φορές μέχρι τώρα, οι καλές επιδόσεις των παράλληλων προγραμμάτων απαιτούν γνώση της οργάνωσης της μηχανής η οποία θα τα φιλοξενήσει.

Δεν πρόκειται να μπούμε σε βάθος στις αρχιτεκτονικές λεπτομέρειες ενός παράλληλου υπολογιστή σε αυτό το κεφάλαιο. Ενδιαφερόμαστε προς το παρόν, κυρίως, για εκείνο το «επικοινωνούν και συνεργάζονται ...» που αναφέρθηκε στον ορισμό των παράλληλων υπολογιστών. Και αυτό είναι ακριβώς που ξεχωρίζει τόσο τους παράλληλους από τους σειριακούς υπολογιστές, όσο και τους παράλληλους υπολογιστές μεταξύ τους. Βασική επομένως παράμετρος είναι το επικοινωνιακό σύστημα του υπολογιστή.

Αρχιτεκτονικά, με βάση τη διάκριση του M.J. Flynn, οι υπολογιστές κατατάσσονται στις εξής κατηγορίες: SISD (single-instruction single-data), SIMD (single-instruction multiple-data) και MIMD (mutiple-instruction multiple-data). Η πρώτη κατηγορία είναι οι κλασικοί σειριακοί υπολογιστές (Σχ. 1.5(α)) οι οποίοι εκτελούν μία εντολή τη φορά (single-instruction) επάνω σε ένα δεδομένο (single-data). Στην κατηγορία SIMD συναντάμε υπολογιστές οι οποίοι εκτελούν μία εντολή τη φορά, αλλά μπορούν να την εφαρμόσουν ταυτόχρονα σε πολλαπλά δεδομένα (multiple-data). Χαρακτηριστικό παράδειγμα αποτελούν οι λεγόμενοι *επεξεργαστές πίνακα* (Σχ. 1.5(β)). Η κατηγορία MIMD θεωρείται η κατηγορία των καθαρά παράλληλων υπολογιστών οι οποίοι μπορούν και εκτελούν ταυτόχρονα πολλαπλές εντολές, με κάθε μία να ασχολείται με διαφορετικό δεδομένο.

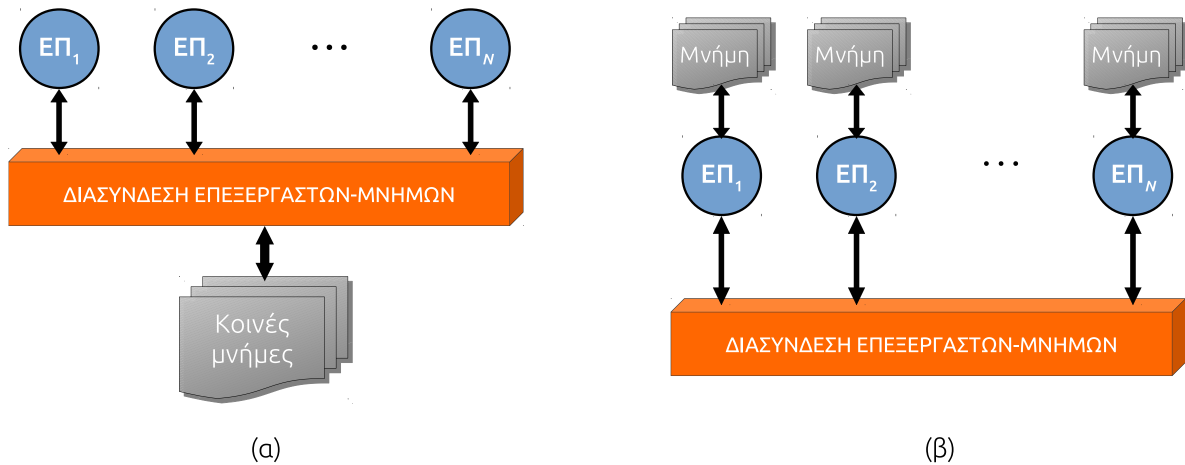
Οι υπολογιστές SIMD αποτελούνται από πολλά επεξεργαστικά στοιχεία (PU) τα οποία συνήθως είναι απλά και εκτελούν απλές πράξεις, όπως πρόσθεση, αφαίρεση κλπ. Η Μονάδα Ελέγχου (CU) είναι αυτή που γνωρίζει ποια είναι η επόμενη εντολή που θα εκτελεστεί και δίνει αυτή την εντολή σε όλα τα επεξεργαστικά στοιχεία. Αυτά όμως, την εκτελούν σε διαφορετικά δεδομένα. Τα δεδομένα τους τα παίρνουν είτε από μία κοινόχρηστη μνήμη είτε από ιδιωτικές μνήμες. Χαρακτηριστικές εφαρμογές είναι οι διανυσματικές πράξεις, π.χ. η πρόσθεση δύο διανυσμάτων A και B με N στοιχεία το καθένα. Η κοινή εντολή που



Σχήμα 1.5 *Αρχιτεκτονικές κατηγορίες υπολογιστών κατά Flynn.*

θα εκτελεστεί είναι η πρόσθεση και τα δεδομένα για τον επεξεργαστή i είναι τα A_i και B_i. Εφόσον διαθέτουμε N PU, σε ένα μόλις βήμα θα έχουμε το επιθυμητό αποτέλεσμα. Μέχρι πριν λίγα χρόνια, τα συστήματα αυτά είχαν περισσότερο ιστορική αξία, καθώς μεσουράνησαν από τη δεκαετία του 1970 μέχρι τα μέσα της δεκαετίας του 1990. Η σύγχρονη εκδοχή τους, όμως, συναντάται στις μονάδες επεξεργασίας γραφικών (GPU) και είναι εξαιρετικά δημοφιλής.

Η γενικότερη μορφή παραλληλίας βρίσκεται στις αρχιτεκτονικές MIMD (Σχ. 1.5(γ)). Τα συστήματα αυτά διαθέτουν N ανεξάρτητους επεξεργαστές όπου ο καθένας μπορεί να εκτελεί διαφορετικές εντολές (multiple-instruction) σε διαφορετικά δεδομένα (multiple-data). Οι υπολογιστές MIMD, οι οποίοι είναι γνωστοί και ως *πολυεπεξεργαστές* (multiprocessors),



Σχήμα 1.6 *Οργανώσεις πολυεπεξεργαστών.*

ανάλογα με το επικοινωνιακό τους σύστημα χωρίζονται σε συστήματα *κοινόχρηστης μνήμης* και *κατανεμημένης μνήμης*.

Στους πολυεπεξεργαστές κοινόχρηστης μνήμης (shared-memory multiprocessors) *δεν υπάρχει απευθείας σύνδεση μεταξύ των επεξεργαστών, παρά μόνο ανάμεσα στους επεξεργαστές και τη μνήμη* η οποία είναι άμεσα προσπελάσιμη από όλους (Σχ. 1.6(α)). Κάθε επεξεργαστής μπορεί να εκτελεί διαφορετική εντολή επάνω σε διαφορετικά δεδομένα τα οποία όμως, μπορούν να τα προσπελάσουν όλοι. Αυτό έχει ως αποτέλεσμα ότι όλοι οι επεξεργαστές «βλέπουν» την ίδια μνήμη, με τον ίδιο τρόπο. Προγραμματιστικά, η κατάσταση δεν διαφέρει πολύ από το γνώριμο σειριακό μοντέλο. Η επικοινωνία μεταξύ των επεξεργαστών γίνεται μέσω της τροποποίησης κοινόχρηστων μεταβλητών στη μνήμη.

Αντίθετα με τους πολυεπεξεργαστές κοινόχρηστης μνήμης, στα συστήματα κατανεμημένης μνήμης (distributed memory multiprocessors—Σχ. 1.6(β)) κάθε επεξεργαστής έχει την ιδιωτική του μνήμη την οποία μπορεί να προσπελάσει απευθείας μόνο αυτός. Η επικοινωνία των επεξεργαστών είναι εφικτή *λόγω ύπαρξης διασυνδετικού δικτύου διά μέσω του οποίου γίνεται ανταλλαγή μηνυμάτων*. Έτσι, οτιδήποτε χρειαστεί ο επεξεργαστής i από τον επεξεργαστή j (π.χ. να προσπελάσει κάποιο δεδομένο από τη μνήμη του j), θα το ζητήσει μέσω μηνύματος από τον j, ο οποίος πάλι μέσω μηνύματος που θα ταξιδέψει στο δίκτυο, θα το στείλει στον επεξεργαστή i.

Τέλος, μία τρίτη, μοιραία κατηγορία είναι οι λεγόμενοι *ομαδοποιημένοι πολυεπεξεργαστές*, οι οποίοι αποτελούν «διασταύρωση» των πολυεπεξεργαστών κοινόχρηστης και κατανεμημένης μνήμης. Πιο συγκεκριμένα, τα συστήματα είναι οργανωμένα όπως στο Σχ. 1.6(β) όπου, όμως, κάθε επεξεργαστής έχει αντικατασταθεί από ομάδα επεξεργαστών οι οποίοι μοιράζονται μία τοπική κοινόχρηστη μνήμη. Το δίκτυο διασύνδεσης πλέον, είναι δίκτυο διασύνδεσης *ομάδων*. Η οργάνωση αυτή φαίνεται να επικρατεί καθολικά σε μετρίου και μεγάλου μεγέθους συστήματα, ειδικά με την έλευση των πολυπύρηνων επεξεργαστών,

όπου μία ομάδα είναι ένας μικρός αριθμός επεξεργαστών, καθένας με πολλαπλούς πυρήνες.

1.4 Προγραμματισμός των παράλληλων υπολογιστών

Ένα αδιαμφισβήτητο γεγονός είναι ότι οι παράλληλες μηχανές εξελίχθηκαν πολύ γρήγορα, ή πιο σωστά, εξελίχθηκαν πολύ πιο γρήγορα απ' ότι ο παράλληλος προγραμματισμός. Βρισκόμαστε σε ένα σημείο όπου υπάρχει πληθώρα αρχιτεκτονικών, καθώς και υλοποιημένων συστημάτων, αλλά ακόμη δεν υπάρχει γενική απάντηση στο ερώτημα: πώς τα προγραμματίζουμε; Είναι βασικό ότι θέλουμε να εκμεταλλευτούμε την περίσσεια των επεξεργαστών στο έπακρο, και όχι να φτάσουμε να χρησιμοποιούμε έναν μόνο από αυτούς—σαν να προγραμματίζουμε έναν σειριακό υπολογιστή. Υπάρχει βέβαια και η συνηθισμένη (και όντως απαραίτητη) περίπτωση όπου ενδιαφερόμαστε ο υπολογιστής να εκτελεί πολλά προγράμματα ταυτόχρονα, με κάθε επεξεργαστή να εκτελεί (σειριακά) διαφορετικό πρόγραμμα. Όμως, όταν μιλάμε για παράλληλο προγραμματισμό, εννοούμε ένα πρόγραμμα, το οποίο χρησιμοποιεί ταυτόχρονα πολλούς επεξεργαστές.

Το πρόβλημα με τον παράλληλο προγραμματισμό ξεκινάει από το γεγονός ότι, το σύνολο των υπαρχόντων προγραμμάτων, καθώς και η όλη προγραμματιστική μεθοδολογία βασίζονται στην υπόθεση ότι από πίσω βρίσκεται μία σειριακή αρχιτεκτονική: οι εντολές εκτελούνται η μία μετά την άλλη και η επόμενη προαπαιτεί να έχει εκτελεστεί η προηγούμενη. Καταλαβαίνουμε λοιπόν, ότι ο προγραμματισμός των παράλληλων υπολογιστών απαιτεί σημαντικά διαφορετική προσέγγιση, καθώς πλέον η σειριακή εκτέλεση μιας σειράς εντολών απέχει πολύ από το ζητούμενο.

Η μετάβαση από τον σειριακό στον παράλληλο προγραμματισμό δεν είναι η απλούστερη υπόθεση και αυτό γιατί, αντίθετα με τον σειριακό προγραμματισμό,

- η ανάπτυξη ενός παράλληλου προγράμματος εξαρτάται λίγο ή πολύ από την αρχιτεκτονική του συστήματος που θα φιλοξενήσει το πρόγραμμα,
- δεν υπάρχει πάντα πλήρης συμβατότητα μεταξύ προγραμμάτων για διαφορετικά μηχανήματα και
- υπάρχουν αρκετά διαφορετικά μοντέλα προγραμματισμού (καθώς και ισάριθμοι τρόποι σκέψης).

Όλα αυτά δυσχεραίνουν τη θέση του προγραμματιστή, σε σημείο που κάποιοι είχαν φτάσει να αμφισβητήσουν ακόμα και τη χρησιμότητα των παράλληλων υπολογιστών. Ένα ακόμη σημαντικό σημείο είναι ότι, ίσως, χρειαστεί επαναπροσδιορισμός των προδιαγραφών του εξεταζόμενου προβλήματος και νέα προσέγγιση στη λύση του. Και αυτό διότι τίποτε δεν μας εξασφαλίζει ότι ο καλύτερος σειριακός αλγόριθμος, μετά από παραλληλοποίηση, δίνει τον καλύτερο παράλληλο αλγόριθμο.

Η ανάγκη για διευκόλυνση στον προγραμματισμό των παράλληλων μηχανών οδήγησε τους σχεδιαστές μεταφραστών σε μια προσπάθεια για αυτόματη δημιουργία παράλληλων προγραμμάτων από σειριακά: ο προγραμματιστής θα είναι σε θέση να γράφει τον κώδικά του σε μια οικεία γλώσσα προγραμματισμού (σειριακή) και ο μεταφραστής θα αναλύσει τον κώδικα, θα ανακαλύψει τμήματά του τα οποία δεν έχουν εξάρτηση μεταξύ τους (και άρα μπορούν να εκτελεστούν παράλληλα) και θα αναλάβει τη διαμοίραση των εργασιών στους διάφορους επεξεργαστές. Αυτού του είδους ο παράλληλος προγραμματισμός ονομάζεται *έμμεσος* (implicit), καθώς ο προγραμματιστής δεν επεμβαίνει καθόλου στη διαδικασία της παραλληλοποίησης.

Η προσπάθεια αυτή κατά το μεγαλύτερο μέρος της δεν απέδωσε τα αναμενόμενα αν και, ακόμα, αποτελεί αντικείμενο έρευνας. Το σημείο στο οποίο σημειώθηκαν οι περισσότερες επιτυχίες, οι οποίες έχουν ήδη βρει τον δρόμο τους σε διάφορα προγραμματιστικά εργαλεία και γλώσσες προγραμματισμού, είναι η αυτόματη παραλληλοποίηση τυποποιημένων βρόχων `for`. Βρόχοι από εντολές `for` είναι βασικές δομές σε οποιοδήποτε πρόγραμμα, αλλά εμφανίζονται σχεδόν πάντα σε μαθηματικά προγράμματα, όπως π.χ. για την επίλυση εξισώσεων. Η μεθοδολογία που έχει αναπτυχθεί είναι σε θέση να παραλάβει αρκετά πολύπλοκους και εμφωλευμένους βρόχους `for` και να τους μετασχηματίσει σε ισοδύναμο κώδικα, ο οποίος όμως έχει εκθέσει τις όποιες δυνατότητες παραλληλοποίησης (δηλαδή ταυτόχρονης εκτέλεσης εντολών) υπάρχουν.

Το τελικό αποτέλεσμα όμως, είναι ότι η γενική μεθοδολογία για την ανάπτυξη ενός παράλληλου προγράμματος επαφίεται ουσιαστικά εξολοκλήρου στον προγραμματιστή. Μιλάμε επομένως, για *άμεσα διατυπωμένο* (explicit) παράλληλο προγραμματισμό όπου ο προγραμματιστής έχει προσδιορίσει τα σημεία στα οποία μπορεί να υπάρξει ταυτόχρονη εκτέλεση εντολών και μάλιστα, ίσως έχει επέμβει και στον τρόπο που αυτές θα διαμοιραστούν και θα επικοινωνήσουν. Με τον άμεσα διατυπωμένο παράλληλο προγραμματισμό, θα ασχοληθούμε στο βιβλίο αυτό.

1.4.1 Τα κύρια μοντέλα παράλληλου προγραμματισμού

Όταν λέμε *μοντέλα προγραμματισμού* (programming models) εννοούμε τον τρόπο με τον οποίο ο προγραμματιστής αντιλαμβάνεται τη λειτουργία του υπολογιστή. Είναι προφανές ότι κάτι τέτοιο εξαρτάται από την αρχιτεκτονική του υπολογιστή, αν και είναι δυνατόν μια συγκεκριμένη μηχανή να υποστηρίζει διαφανώς, παραπάνω από ένα διαφορετικά προγραμματιστικά μοντέλα.

Τρία είναι τα βασικά μοντέλα στον παράλληλο προγραμματισμό, τα οποία αποτελούν κατά κάποιο τρόπο μία αφαίρεση των οργανώσεων που είδαμε στην Ενότητα 1.3. Το πρώτο μοντέλο, είναι το λεγόμενο μοντέλο *παραλληλισμού δεδομένων* (data-parallel model). Στο μοντέλο αυτό, ο προγραμματιστής έχει στη διάθεσή του N επεξεργαστές, οι οποίοι σε κάθε χρονική στιγμή εκτελούν την ίδια ακριβώς εντολή, σε διαφορετικά πιθανώς

δεδομένα. Στο μοντέλο αυτό επομένως, χρειάζονται προγραμματιστικές δομές που διαμοιράζουν στους επεξεργαστές τα δεδομένα, επάνω στα οποία θα εκτελεστεί η κοινή εντολή. Είναι φανερό ότι το μοντέλο αυτό, απευθύνεται σε προβλήματα που εμπεριέχουν διανυσματικούς υπολογισμούς, ενώ αρχιτεκτονικά ταιριάζει σε συστήματα τύπου SIMD. Μπορούν πάντως να το υποστηρίξουν εύκολα και συστήματα MIMD. Όμως, η αυστηρά συγχρονισμένη εκτέλεση των εντολών σε ένα τέτοιο μοντέλο, έχει ως αποτέλεσμα μη αποδοτικό κώδικα για συστήματα MIMD, αφού είναι απαραίτητος ο καθολικός συγχρονισμός των επεξεργαστών μετά από κάθε εντολή. Η λύση είναι να απαλλαγούμε από την ανάγκη αυτή του συγχρονισμού. Καταλήγουμε, έτσι, στο λεγόμενο μοντέλο SPMD (single-program multiple-data) όπου οι επεξεργαστές δεν εκτελούν την ίδια εντολή αλλά το ίδιο κομμάτι κώδικα, με τον δικό του ρυθμό ο καθένας, σε διαφορετικά δεδομένα. Συγχρονισμός απαιτείται μόνο αν υπάρξει ανάγκη επικοινωνίας μεταξύ των επεξεργαστών.

Τα άλλα δύο μοντέλα υποθέτουν τον λεγόμενο *παραλληλισμό ελέγχου* (control parallelism) όπου πλέον στον προγραμματιστή προσφέρονται επεξεργαστές που δεν είναι αναγκασμένοι να εκτελούν τον ίδιο κώδικα, αλλά μπορούν να εκτελούν διαφορετικές εντολές με διαφορετικά δεδομένα. Τα μοντέλα αυτά ταιριάζουν σε μηχανήματα MIMD και είναι με διαφορά τα πιο δημοφιλή.

Στο μοντέλο *κοινόχρηστου χώρου διευθύνσεων* (shared address space model) ο προγραμματιστής σχεδιάζει το πρόγραμμά του ως ένα σύνολο διεργασιών, οι οποίες προσπελαύνουν μια συλλογή από κοινόχρηστες μεταβλητές. Φυσιολογικά, το μοντέλο αυτό ταιριάζει απόλυτα σε έναν πολυεπεξεργαστή κοινόχρηστης μνήμης όπου οι καθολικές μεταβλητές αποθηκεύονται στην κοινή μνήμη και προσπελαύνονται από όλους. Αυτός όμως, είναι και ο λόγος για τον οποίο υπάρχει περίπτωση απροσδιόριστων καταστάσεων, όταν π.χ. παραπάνω από μία διεργασίες προσπαθούν να αλλάξουν ταυτόχρονα την τιμή μίας κοινόχρηστης μεταβλητής. Επομένως, στο μοντέλο αυτό παρέχονται τρόποι και δομές που εμποδίζουν τις ταυτόχρονες προσπελάσεις κοινόχρηστων μεταβλητών, οι οποίες οδηγούν σε ανεπιθύμητες καταστάσεις. Παράδειγμα τέτοιων δομών αποτελούν οι γνωστοί, κλασικοί σηματοφόροι (semaphores).

Στο μοντέλο *μεταβίβασης μηνυμάτων* (message-passing model), το πρόγραμμα είναι μια συλλογή αυτόνομων διεργασιών, οι οποίες δεν έχουν καμία κοινή μεταβλητή και οι οποίες έχουν τη δυνατότητα να επικοινωνούν μεταξύ τους ανταλλάσσοντας δεδομένα. Επομένως, υπάρχουν προγραμματιστικές δομές που καθιστούν δυνατή την αποστολή και λήψη μηνυμάτων μεταξύ των διεργασιών. Είναι λογικό ότι υπάρχει άμεση αντιστοιχία με τους πολυεπεξεργαστές κατανεμημένης μνήμης.

Αξίζει να σημειωθεί ότι ένας υπολογιστής που διαθέτει αρχιτεκτονική κατανεμημένης μνήμης, μπορεί να υποστηρίξει προγραμματισμό τύπου κοινόχρηστου χώρου διευθύνσεων μέσω οργάνωσης, γνωστής ως *κατανεμημένη κοινή μνήμη* (distributed shared memory). Ακόμα πιο εύκολα μπορεί να γίνει το αντίστροφο, δηλαδή, μια μηχανή κοινόχρηστης μνήμης να υποστηρίζει προγράμματα γραμμένα με βάση το μοντέλο μεταβίβασης μηνυμά-

των. Για τον λόγο αυτό, θα πρέπει να τονιστεί ότι η οργάνωση του υπολογιστή δεν είναι πάντα η ίδια με αυτή που μας προδιαθέτει το προγραμματιστικό μοντέλο (αν και συνήθως τα βέλτιστα προγράμματα είναι αυτά που χρησιμοποιούν το μοντέλο που συμφωνεί με την αρχιτεκτονική του συστήματος). Το δεύτερο, βέβαια, είναι αυτό που μας ενδιαφέρει ως προγραμματιστές.

1.5 Βασική μεθοδολογία παραλληλοποίησης

Αν και υπάρχουν ορισμένοι κανόνες οι οποίοι μπορούν να εφαρμοστούν σε αρκετές περιπτώσεις, ο παράλληλος προγραμματισμός παραμένει ένα είδος τέχνης, καθώς η ποιότητα ενός παράλληλου προγράμματος εξαρτάται από την εμπειρία και την ικανότητα του προγραμματιστή, όπως και από τις επιδόσεις του συστήματος που θα φιλοξενήσει το πρόγραμμα.

Εμείς εδώ, θα δούμε ορισμένες από τις τυποποιημένες μεθόδους που εφαρμόζονται στην πράξη. Οι συγκεκριμένες μέθοδοι αποτελούνται ουσιαστικά από δύο βήματα: τον *διαχωρισμό* (partitioning) του προγράμματος σε παράλληλα τμήματα και την *τοποθέτησή* τους στους επεξεργαστές. Το πρώτο βήμα είναι σε μεγάλο ποσοστό ανεξάρτητο από την αρχιτεκτονική του συστήματος, ενώ το δεύτερο είναι συνυφασμένο με αυτή.

Ο διαχωρισμός σε παράλληλα τμήματα μπορεί να χωριστεί σε δύο φάσεις: τη *διάσπαση* (decomposition) σε εργασίες και την *ανάθεση* των εργασιών (assignment) σε εκτελεστικές οντότητες (διεργασίες ή νήματα, όπως θα δούμε αργότερα). Η διάσπαση του προβλήματος σε *εργασίες* (tasks), είναι ίσως το βασικότερο βήμα στην παραλληλοποίηση. Ο όρος «εργασίες» δεν έχει κάποιο αυστηρό ορισμό: είναι γενικά κάποια τμήματα του υπολογισμού τα οποία μπορούν να εκτελεστούν παράλληλα. Μία εργασία εκτελείται εξ ολοκλήρου από έναν επεξεργαστή. Για παράδειγμα, στο πρόβλημα των N σωμάτων (Πρόγραμμα 1.4) μία εργασία θα μπορούσε να είναι η εντολή:

```
        F[i] = add( F[i], f(i,j) );
```

(οπότε προκύπτουν συνολικά N^2 εργασίες για κάθε επανάληψη του βρόχου t), η οποία συνεισφέρει τη δύναμη $\mathbf{f}_{ij}$ στη συνολική δύναμη που ασκείται στο i-οστό σώμα. Θα μπορούσε, κάλλιστα, να είναι και το τμήμα του κώδικα:

```
      for (j = 0; j < N; j++)
        F[i] = add( F[i], f(i,j) );
```

(οπότε θα υπάρχουν N τέτοιες εργασίες), το οποίο αναλαμβάνει ολόκληρο τον υπολογισμό της συνολικής δύναμης που ασκείται στο σώμα i.

Η επιλογή των ατομικών αυτών εργασιών είναι αυτό που καθορίζει τον κόκκο παραλληλίας που αναφέραμε στην Ενότητα 1.1. Στο παράδειγμά μας, η πρώτη περίπτωση της μίας εντολής ανά εργασία δίνει λεπτό κόκκο παραλληλίας, ενώ η δεύτερη επιλογή είναι

πιο χονδρόκοκκη. Η εξαγωγή των βασικών εργασιών από τον σειριακό κώδικα επηρεάζει άμεσα τον βαθμό παραλληλίας που θα επιτύχουμε, δηλαδή το πόσο καλά μπορούμε να χωρίσουμε το πρόβλημα σε κατάλληλο αριθμό τμημάτων, ταυτόχρονα εκτελέσιμων, ώστε να εκμεταλλευτούμε τις δυνατότητες που μας παρέχει ο παράλληλος υπολογιστής.

Η *διεργασία* (process) ή το *νήμα* (thread) είναι μια συλλογή από εργασίες και εκτελείται σειριακά. Θα χρησιμοποιήσουμε τον όρο *εκτελεστικές οντότητες* για να αναφερόμαστε γενικά είτε σε διεργασίες είτε σε νήματα. Μια εκτελεστική οντότητα μπορεί, για παράδειγμα, να είναι μια διαδικασία (procedure) του σειριακού κώδικα ή πιο συνηθισμένα, μία κατά περίπτωση επιλογή από σχετιζόμενες εργασίες (που π.χ. εργάζονται επάνω στις ίδιες μεταβλητές). Η ανάθεση (ομαδοποίηση) εργασιών σε εκτελεστικές οντότητες είναι πολλές φορές απαραίτητη, όταν υπάρχουν πολλές εργασίες αλλά λίγοι επεξεργαστές. Έτσι, αν στο παράδειγμα των N σωμάτων διαθέτουμε P επεξεργαστές, μπορούμε να αποφασίσουμε το χώρισμα της εκτέλεσης σε P διεργασίες. Στην κάθε διεργασία ανατίθενται N/P εργασίες που υπολογίζουν τις συνολικές δυνάμεις σε N/P σώματα. Με αυτόν τον τρόπο, η k-οστή διεργασία θα περιλάμβανε τις εργασίες που υπολογίζουν συνολικές δυνάμεις για τα σώματα $(k-1)\frac{N}{P}$, $(k-1)\frac{N}{P}+1$, ...:

```
Process_k:
{
  for (i = (k-1)*N/P; i < k*N/P; i++) {
    F[i] = zero();
    for (j = 0; j < i; j++)
      F[i] = add( F[i], f(i,j) );
  }
}
```

Γενικά, μια σωστή ανάθεση θα πρέπει να μοιράζει τις εργασίες με τέτοιο τρόπο, ώστε:

(α) οι εκτελεστικές οντότητες να εκτελούν παρόμοιο σε όγκο υπολογισμό, και άρα οι επεξεργαστές να έχουν παρόμοιο φόρτο εργασίας. Σκοπός, δηλαδή, είναι η *ισοκατανομή του φόρτου* (load balancing), στοιχείο σημαντικό για την επίτευξη υψηλών επιδόσεων κατά την εκτέλεση του προγράμματος,

(β) να μειώνεται η ανάγκη επικοινωνίας μεταξύ των οντοτήτων (και άρα των επεξεργαστών που θα τις εκτελέσουν), αναθέτοντας συνεργαζόμενες εργασίες στην ίδια εκτελεστική οντότητα.

Η *τοποθέτηση* των εκτελεστικών οντοτήτων, το δεύτερο τμήμα στην παραλληλοποίηση, εξαρτάται άμεσα από την αρχιτεκτονική του υπολογιστή και το προγραμματιστικό μοντέλο. Η τοποθέτηση μπορεί να χωριστεί σε δύο τμήματα: την *ενορχήστρωση* (orchestration) των οντοτήτων και την *αντιστοίχισή* τους (mapping) στους επεξεργαστές.

Με τον όρο «ενορχήστρωση» εννοείται ο καθορισμός του τρόπου που θα επικοινωνούν και θα συντονίζονται οι διεργασίες ή τα νήματα, κάτι που προφανώς εξαρτάται

από το προγραμματιστικό μοντέλο που θα επιλεγεί. Ως παράδειγμα, αν χρησιμοποιηθεί μεταβίβαση μηνυμάτων, θα πρέπει να διευκρινιστούν τα σημεία στα οποία θα σταλεί ή θα ληφθεί κάποιο μήνυμα και αν τα μηνύματα θα είναι μικρά ή μεγάλα σε μέγεθος (δηλαδή αν είναι συμφέρουσα η αποστολή πολλών δεδομένων μαζί ή όχι). Επίσης, στο βήμα της ενορχήστρωσης καθορίζεται και η σειρά εκτέλεσης. Έτσι, θα πρέπει να εξασφαλιστεί με κάποιο τρόπο (π.χ. με κάποια επικοινωνία) ότι καμία εκτελεστική οντότητα που απαιτεί αποτελέσματα από άλλες, δεν θα αρχίσει να εκτελείται πριν αυτές τελειώσουν. Επαναλαμβάνουμε ότι στο στάδιο αυτό, όλα εξαρτώνται από το προγραμματιστικό μοντέλο και από τις δομές που παρέχει η γλώσσα προγραμματισμού που θα χρησιμοποιηθεί.

Με τον όρο «αντιστοίχιση» εννοείται ο καθορισμός του επεξεργαστή που θα εκτελέσει την κάθε εκτελεστική οντότητα. Η αντιστοίχιση (αλλά και η ενορχήστρωση ως έναν βαθμό) μπορεί να είναι στατική ή δυναμική. Στην πρώτη περίπτωση, ο προγραμματιστής καθορίζει πλήρως την αντιστοίχιση η οποία δεν μεταβάλλεται κατά τη διάρκεια εκτέλεσης του προγράμματος. Στη δεύτερη περίπτωση συνήθως, την αντιστοίχιση την αναλαμβάνει το λειτουργικό σύστημα και κατά τη διάρκεια της εκτέλεσης τοποθετεί οντότητες σε κατάλληλους επεξεργαστές, π.χ. ανάλογα με τον φόρτο εργασίας τους. Επίσης, υπάρχει περίπτωση κατά την διάρκεια της εκτέλεσης να μεταφερθεί μια διεργασία από έναν επεξεργαστή σε έναν άλλο, αν κριθεί απαραίτητο. Κάτι ενδιάμεσο συμβαίνει στα πιο σύγχρονα συστήματα, όπου ο προγραμματιστής υποδεικνύει ορισμένες ιδιότητες της αντιστοίχισης, αλλά το λειτουργικό σύστημα μπορεί δυναμικά να τις αλλάξει προκειμένου να γίνει καλύτερη εκμετάλλευση πόρων.

1.5.1 Διάσπαση σε εργασίες

Όπως τονίσαμε παραπάνω, αυτό είναι ίσως το σημαντικότερο και δυσκολότερο βήμα στον παράλληλο προγραμματισμό και αυτό διότι, από αυτό εξαρτάται αν θα αποκαλυφθούν οι όποιες δυνατότητες παράλληλης εκτέλεσης υπάρχουν. Με το θέμα αυτό θα ασχοληθούμε εκτενέστερα εδώ. Βέβαια, τα υπόλοιπα βήματα είναι επίσης σημαντικά, αφού θα μεταφράσουν τις δυνατότητες παραλληλισμού σε πραγματική αύξηση της ταχύτητας εκτέλεσης σε σχέση με τον σειριακό υπολογισμό. Τα βήματα αυτά είναι που θα εκμεταλλευτούν τους πραγματικούς πόρους του συστήματος σωστά ή που θα καταστρέψουν τα προσόντα που διαθέτει ο αλγόριθμος. Και είναι αρκετές οι δημοσιευμένες περιπτώσεις που, μόνο μετά από επαναληπτική βελτιστοποίηση των τελευταίων βημάτων, έγινε εφικτή η αξιοπρεπής εκτέλεση του κώδικα.

Υπάρχουν διάφορες τεχνικές διάσπασης που έχουν χρησιμοποιηθεί στην πράξη σε πολλά προβλήματα. Σε εφαρμογές όπου εξομοιώνονται φυσικά φαινόμενα, οι δομές δεδομένων που χρησιμοποιούνται ταιριάζουν απόλυτα με το σύστημα υπό μελέτη. Για παράδειγμα, οι θερμοκρασίες στα σημεία ενός χάρτη αναπαριστάνονται φυσιολογικά με έναν δισδιάστατο πίνακα πραγματικών αριθμών. Στην περίπτωση αυτή, η δομή δεδομένων μπο-

ρεί να διασπαστεί ακριβώς όπως υποδιαιρείται το φυσικό σύστημα. Μια τέτοια διάσπαση λέγεται *γεωμετρική*.

Πιο γενική είναι η διάσπαση του *πεδίου ορισμού* (domain decomposition), όπου πάλι υπάρχει μια μεγάλη βασική δομή στην οποία στηρίζεται ο υπολογισμός. Εάν οι υπολογισμοί που εκτελούνται από το σειριακό πρόγραμμα είναι παρόμοιοι για κάθε στοιχείο της δομής, τότε η δομή μπορεί να διασπαστεί σε ίσα τμήματα και κάθε τμήμα να το αναλάβει ξεχωριστή εργασία. Αυτού του είδους η διάσπαση μπορεί να εφαρμοστεί, όπως είδαμε πριν, και στο πρόβλημα των N σωμάτων. Ακριβώς επειδή γίνεται διάσπαση των δεδομένων σε παρόμοια τμήματα όπου εκτελούνται παρόμοιοι υπολογισμοί, πολλές φορές αυτού του είδους η διάσπαση οδηγεί σε προγράμματα τύπου SPMD.

Συνηθισμένη τεχνική είναι και η *επαναληπτική* διάσπαση, ένα είδος δυναμικής διάσπασης σε εργασίες ή διεργασίες. Εφαρμόζεται σε for-βρόχους και για κάθε επανάληψη του βρόχου ορίζει μία ξεχωριστή εργασία, δημιουργώντας έτσι μια ομάδα εργασιών ίσες σε αριθμό με τις επαναλήψεις του βρόχου. Όποιος επεξεργαστής δεν είναι απασχολημένος με κάτι άλλο, μπορεί να παίρνει και να εκτελεί κάποια από αυτές τις εργασίες, ώσπου να εκτελεστούν όλες. Προσέξτε ότι έτσι, όλοι οι επεξεργαστές διατηρούνται απασχολημένοι όλη την ώρα και ως αποτέλεσμα, εξισορροπείται δυναμικά ο φόρτος. Βέβαια, για να έχει νόημα αυτή η τεχνική θα πρέπει να υπάρχουν πολύ περισσότερες εργασίες απ' ό,τι επεξεργαστές.

Η *αναδρομική* διάσπαση ή διάσπαση *ελεγκτή-εργάτη* είναι παρόμοια με την τακτική «Διαίρει και βασίλευε» που είναι γνωστή, π.χ. από τον αλγόριθμο ταξινόμησης quicksort. Ο αρχικός υπολογισμός ανατίθεται σε έναν επεξεργαστή (ελεγκτή) ο οποίος τον χωρίζει σε δύο ή παραπάνω υποτμήματα και τα διανέμει σε ισάριθμους επεξεργαστές (εργάτες). Αυτοί με τη σειρά τους είτε εκτελούν τα υποτμήματα και επιστρέφουν τα αποτελέσματα στον ελεγκτή είτε μπορούν με τη σειρά τους να γίνουν ελεγκτές, και το τμήμα που ανέλαβαν να το χωρίσουν σε υποτμήματα που τα διανέμουν σε άλλους εργάτες κ.ο.κ. Εδώ θα πρέπει να προσεχθεί, ώστε ο ελεγκτής να διαμοιράζει τις εργασίες ισοζυγισμένα σε εργάτες.

Τέλος, σε πολλές περιπτώσεις δεν μπορεί να εφαρμοστεί η διάσπαση του πεδίου ορισμού, γιατί το πεδίο ορισμού και η αντίστοιχη δομή δεδομένων δεν είναι συμμετρικά ή μεταβάλλονται δυναμικά κατά τη διάρκεια της εκτέλεσης. Πιθανώς να μπορεί να διασπασθεί ο υπολογισμός σε διάφορες φάσεις, μέσω των οποίων διαμορφώνεται το τελικό αποτέλεσμα. Για παράδειγμα, οι αλγόριθμοι κατανόησης εικόνας μπορούν να χωριστούν σε μια σειρά φάσεων όπου το αποτέλεσμα της μίας περνά ως δεδομένο στην επόμενη, μέχρι να αναγνωριστεί η σκηνή της εικόνας: βελτίωση της εικόνας, εντοπισμός των ακμών, κατηγοριοποίηση / αναγνώριση προτύπων, ανάλυση κίνησης και σκηνής. Η κάθε φάση μπορεί να δοθεί ως ξεχωριστή διεργασία που όμως για να εκκινηθεί, θα πρέπει να τελειώσουν οι υπόλοιπες που αντιστοιχούν σε προηγούμενες φάσεις. Κάθε φάση την αναλαμβάνει ξεχωριστός επεξεργαστής. Η μέθοδος αυτή, είναι γνωστή ως *λειτουργική* διάσπαση. Το ζήτημα είναι ότι συνήθως δεν μπορούν να ξεχωρίσουν πολλά τέτοια στάδια σε όλα τα προβλήματα, οπότε η

λειτουργική διάσπαση συνδυάζεται με κάποιες άλλες μορφές διάσπασης (π.χ. γεωμετρικές ή επαναληπτικές) για τις επιμέρους φάσεις.

1.6 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις

Με το κεφάλαιο αυτό, μπήκαμε στον χώρο των παράλληλων υπολογιστών, της αρχιτεκτονικής τους και των ιδιαίτερων χαρακτηριστικών του προγραμματισμού τους. Οι υπολογιστές αυτοί είναι ένα φυσικό επακόλουθο της ανάγκης να ξεφύγουμε από το κλασικό σειριακό μοντέλο, το οποίο αποδεικνύεται ανίσχυρο μπροστά στις μεγάλες υπολογιστικές προκλήσεις πολλών επιστημονικών και ανθρωπιστικών προβλημάτων.

Η λίστα «Top500» των 500 κορυφαίων υπολογιστών στον κόσμο [T500], είναι εξαιρετικά ενδιαφέρουσα για κάποιον που θέλει να γνωρίζει τους ισχυρότερους υπερυπολογιστές στον πλανήτη. Πρόκειται για έναν κατάλογο ο οποίος ανανεώνεται κάθε εξάμηνο και κατατάσσει προχωρημένα παράλληλα συστήματα τα οποία χρησιμοποιούνται ενεργά, με βάση τις επιδόσεις τους σε συγκεκριμένα μετροπρογράμματα. Στο άρθρο [Hors12], μπορεί κανείς να βρει περισσότερες λεπτομέρειες σχετικά με την ιστορία της λίστας, τα συστήματα όπως εξελίχθηκαν και τις εφαρμογές τους.

Από την άλλη μεριά, εξίσου ενδιαφέρουσες είναι και οι εφαρμογές που καλούνται να εκμεταλλευτούν την υπολογιστική ισχύ των παραπάνω μηχανών. Γνωστές και ως μεγάλες προκλήσεις, καλύπτουν όλο το φάσμα της επιστήμης. Περιλαμβάνουν προβλήματα από τη φυσική, τη χημεία, τη βιολογία, την ιατρική, κ.α., που έχουν σχέση με την εξέλιξη και την βελτίωση της ζωής της ανθρωπότητας. Ενδεικτικά, μία πολύ ενδιαφέρουσα συλλογή από εφαρμογές αυτού του τύπου, μπορεί κάποιος να βρει στην ιστοσελίδα του Ινστιτούτου Προχωρημένων Προσομοιώσεων (IAS) του Υπερυπολογιστικού Κέντρου του Jülich (JSC)[3].

Η χρήση δύο ή παραπάνω επεξεργαστών έχει αντίκτυπο και στο σχεδιασμό αλλά και στον προγραμματισμό των παράλληλων συστημάτων, καθώς πλέον απαιτούνται τρόποι για:

- το «κομμάτιασμα» των προβλημάτων και την ανάθεση των τμημάτων σε διαφορετικούς επεξεργαστές,
- την επικοινωνία των επεξεργαστών προκειμένου να λυθεί το υπολογιστικό πρόβλημα.

Τα δύο αυτά στοιχεία οδήγησαν από τη μία σε διαφορετικού τύπου αρχιτεκτονικές (SIMD, MIMD κοινόχρηστης / κατανεμημένης μνήμης) και από την άλλη σε διαφορετικά προγραμματιστικά μοντέλα (παραλληλισμού δεδομένων, κοινού χώρου διευθύνσεων, μεταβίβασης μηνυμάτων). Τόσο τις αρχιτεκτονικές όσο και τα προγραμματιστικά μοντέλα αυτά, θα τα

[3] http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/_node.html

δούμε με λεπτομέρεια στα κεφάλαια που ακολουθούν. Η ιστορική διάκριση των υπολογιστικών συστημάτων σε SISD, SIMD και MIMD, έγινε από τον M.J. Flynn το 1972 [Flyn72].

Αν και κάθε προγραμματιστικό μοντέλο έχει τα δικά του χαρακτηριστικά και απαιτεί τον δικό του ιδιαίτερο τρόπο σκέψης [Dima13], η διαδικασία για παραλληλοποίηση ενός σειριακού αλγόριθμου θα περάσει γενικά από τέσσερα στάδια: τη διάσπασή του σε εργασίες, την ανάθεση των εργασιών σε εκτελεστικές οντότητες, την ενορχήστρωση των τελευταίων και την αντιστοίχισή τους στους επεξεργαστές για να τις εκτελέσουν.

Η ξενόγλωσση βιβλιογραφία για παράλληλα συστήματα και τον προγραμματισμό τους είναι πολύ πλούσια και περιλαμβάνει γενικά βιβλία αλλά και βιβλία που ασχολούνται μόνο με το κομμάτι της αρχιτεκτονικής ή μόνο με το κομμάτι του προγραμματισμού. Ενδεικτικά αναφέρεται το [GGKK03] ως ένα πολύ καλό και πλήρες σύγγραμμα, αν και κάπως παλαιότερο, το οποίο καλύπτει τα περισσότερα θέματα που αφορούν παράλληλα συστήματα. Η ελληνική βιβλιογραφία για παράλληλους υπολογιστές είναι μάλλον περιορισμένη. Σημειώνουμε τα αξιόλογα βιβλία των Πάντζιου, Μάμαλη και Τομάρα [PMT13] και Παπαδάκη και Διαμαντάρα [PaDi12], τα οποία πραγματεύονται αρκετά από τα σημεία που θα καλύψουμε κι εμείς εδώ.

Η μελέτη των παράλληλων υπολογιστών και του προγραμματισμού τους, αποτελεί ένα συναρπαστικό αντικείμενο το οποίο μόλις πρόσφατα έγινε *άκρως απαραίτητο* για οποιονδήποτε μελετά την επιστήμη και τη μηχανική των ηλεκτρονικών υπολογιστών. Πρόκειται για ένα μοναδικό μείγμα υλικού, λογισμικού, εφαρμογών και θεωρίας το οποίο τελικά «δουλεύει» στην πράξη με θεαματικά αποτελέσματα [FWM94].

Προβλήματα

1.1 – Στο πρόγραμμα που παραλληλοποιήσαμε στην Ενότητα 1.2.1, όπως σημειώσαμε, χρειάζεται συντονισμός και συνεργασία σε κάποιο σημείο. Μήπως απαιτείται και δεύτερο σημείο συγχρονισμού, και αν ναι, πού;

1.2 – Μια δημοφιλής μέθοδος για παράλληλους υπολογισμούς είναι η χρήση πολλαπλών σταθμών εργασίας ή προσωπικών υπολογιστών που ήδη συνδέονται σε ένα τοπικό δίκτυο, αντί για την αγορά ακριβών παράλληλων υπολογιστών. Αν μετρήσει κανείς τις ταχύτητες επικοινωνίας σε ένα τέτοιο δίκτυο (π.χ. με την εντολή `ping` του UNIX) θα δει ότι μικρά μηνύματα απαιτούν χρόνο μεταφοράς της τάξης του 1 msec.

(α) Υπολογίστε σε αυτό το χρονικό διάστημα πόσες εντολές εκτελεί ένας απλός επεξεργαστής (σήμερα ακόμα και οι πιο ανίσχυροι επεξεργαστές φτάνουν τις

1. Χονδρός κόκκος παραλληλίας
2. Προφανής παραλληλισμός
3. Υπολογιστές SIMD
4. Μαζικά παράλληλοι υπολογιστές
5. Έμμεσος παράλληλος προγραμματισμός
6. SPMD
7. Μοντέλο μεταβίβασης μηνυμάτων
8. Διαχωρισμός υπολογιστικών οντοτήτων
9. Επαναληπτική διάσπαση
10. Αναδρομική διάσπαση

(α) Προγραμματιστικό μοντέλο όπου όλοι οι επεξεργαστές εκτελούν το ίδιο κομμάτι κώδικα.

(β) Διάσπαση ελεγκτή-εργάτη.

(γ) Διάσπαση και ανάθεση.

(δ) Με χιλιάδες επεξεργαστές.

(ε) Αυτόματη παραλληλοποίηση σειριακών προγραμμάτων.

Πίνακας 1.2 *Έννοιες και ορισμοί για το Πρόβλημα 1.3.*

100.000.000 εντολές το δευτερόλεπτο).

(β) Από την απάντηση στο προηγούμενο ερώτημα, τι θα λέγατε για τον κόκκο παραλληλίας των προγραμμάτων που υλοποιούνται σε τέτοια συστήματα;

1. Σε ποια κατηγορία του Flynn ταιριάζει καλύτερα ένα τέτοιο σύστημα;

1.3 – Στον Πίνακα 1.2, ποιες πέντε από τις έννοιες στην αριστερή στήλη συνδέονται με τους ορισμούς στη δεξιά στήλη;

2 Οργάνωση Κοινόχρηστης Μνήμης

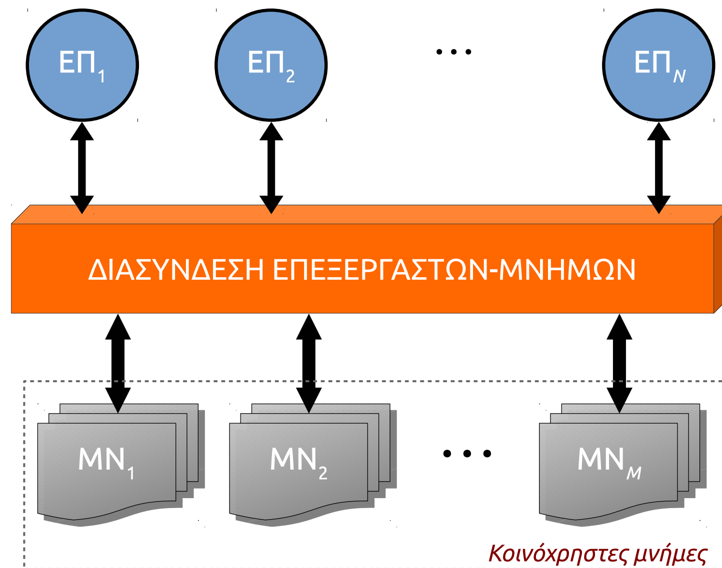
Στο κεφάλαιο αυτό θα ασχοληθούμε με τους πολυεπεξεργαστές κοινόχρηστης μνήμης, γνωστούς και απλά ως *πολυεπεξεργαστές*. Θα δούμε βασικά σημεία της οργάνωσής τους και κυρίως, θα εμβαθύνουμε στο δίκτυο διασύνδεσης επεξεργαστών-μνημών. Το δίκτυο αυτό αποτελεί ίσως, το σημαντικότερο κρίκο της αρχιτεκτονικής και καθορίζει σε μεγάλο βαθμό την απόδοση του συστήματος. Επίσης, θα δούμε το σημαντικό πρόβλημα της συνοχής της κρυφής μνήμης, το οποίο αποτελεί αναπόσπαστο κομμάτι της αρχιτεκτονικής αυτών των υπολογιστών, ενώ θα μελετήσουμε και το μοντέλο συνέπειας που εγγυάται η μνήμη τους. Τέλος, θα γνωρίσουμε την οργάνωση των πολυπύρηνων επεξεργαστών και την ιεραρχία της μνήμης σε αυτούς.

Οι υπολογιστές MIMD κοινόχρηστης μνήμης ήταν ανέκαθεν τα πιο ευρέως διαδεδομένα (και εμπορικά επιτυχημένα) παράλληλα συστήματα. Είναι, μάλιστα, χαρακτηριστικό ότι η αρχιτεκτονική τους έχει εισχωρήσει πλέον και στην αγορά των οικονομικών οικιακών συστημάτων τύπου PC είτε με τη μορφή των διπύρηνων / τετραπύρηνων επεξεργαστών είτε με μητρικές κάρτες συστήματος που δέχονται δύο ή τέσσερις επεξεργαστές. Στο κεφάλαιο αυτό, αρχικά θα υποθέσουμε ότι όταν μιλάμε για επεξεργαστή εννοούμε μονοπύρηνο, δηλαδή με έναν επεξεργαστικό πυρήνα. Στην Ενότητα 2.8, θα δούμε τι αλλάζει με τους πολυπύρηνους επεξεργαστές.

Το κόστος των συστημάτων αυτών μπορεί να είναι από πολύ χαμηλό έως και πολύ υψηλό, ανάλογα με τη δομή που χρησιμοποιείται για τη διασύνδεση των επεξεργαστών και της κοινής τους μνήμης. Τα οικονομικά συστήματα, όπως θα δούμε στην Ενότητα 2.1, βασίζονται στη διασύνδεση με έναν απλό δίαυλο. Ένας τουλάχιστον δίαυλος υπάρχει ούτως ή άλλως σε κάθε σειριακό υπολογιστικό σύστημα προκειμένου να επικοινωνεί ο επεξεργαστής με την κύρια μνήμη και τις συσκευές εισόδου / εξόδου. Επιπρόσθετα, οι δίαυλοι είναι λίγο έως πολύ καθιερωμένης και τυποποιημένης αρχιτεκτονικής (όπως οι γνωστοί δίαυλοι ISA και PCI). Ως αποτέλεσμα, δεν φαίνεται δύσκολο κανείς να προσθέσει μερικούς επιπλέον επεξεργαστές χωρίς να προκαλέσει, θεωρητικά τουλάχιστον, καμία τροποποίηση στο σύστημα.

Η βασική αυτή ιδέα είναι πολύ σωστή και έχει εφαρμοστεί με επιτυχία, όπως δείχνει η πράξη. Με μία μικρή λεπτομέρεια, όμως: να μην προστεθούν πάρα πολλοί επεξεργαστές, διότι οι επιδόσεις πέφτουν σημαντικά. Έτσι, αν ο σκοπός μας είναι ο σχεδιασμός μίας μεγάλης παράλληλης μηχανής με κοινόχρηστη μνήμη, η ιδέα του διαύλου θα πρέπει να εγκαταλειφθεί για τις πιο πολύπλοκες δομές των δικτύων με διακόπτες. Αρκετά από αυτά τα δίκτυα θα τα δούμε στην Ενότητα 2.2.

Ανεξάρτητα, όμως, από ποιο δίκτυο επεξεργαστών-μνημών θα χρησιμοποιηθεί, ουσιαστική επιτάχυνση στη λειτουργία του συστήματος επιτυγχάνεται μόνο με τη χρήση κρυφής μνήμης (cache)—ξεχωριστή σε κάθε επεξεργαστή—ώστε να αποφεύγεται όσο το δυνατόν περισσότερο η χρονοβόρα προσπέλαση της κύριας μνήμης. Την ευεργετική χρήση της κρυφής μνήμης θα την δούμε στην Ενότητα 2.4, όπου επίσης θα ανακαλύψουμε ένα σημαντικό πρόβλημα που αυτή δημιουργεί, το πρόβλημα της συνοχής. Οι Ενότητες 2.5 και 2.6, ασχολούνται με λύσεις στο πρόβλημα της συνοχής της κρυφής μνήμης, οι οποίες εξαρτώνται από το δίκτυο μεταξύ επεξεργαστών και μνημών. Στην Ενότητα 2.5, θα δούμε τα λεγόμενα πρωτόκολλα παρακολούθησης, τα οποία χρησιμοποιούνται στην περίπτωση που το σύστημα βασίζεται σε δίαυλο. Λύσεις αυτής της μορφής έχουν ενσωματωθεί σε πολλούς εμπορικούς επεξεργαστές, ώστε να αποτελούν αυτόνομες και ολοκληρωμένες μονάδες, έτοιμες για πολυεπεξεργαστικά συστήματα. Στην Ενότητα 2.6, τέλος, θα δούμε τα λεγόμενα πρωτόκολλα καταλόγων, τα οποία χρησιμοποιούνται κυρίως σε συστήματα που δεν βασίζονται σε διασύνδεση διαύλου. Είναι, επίσης, απαραίτητα σε πολυεπεξεργαστές κατανεμημένης κοινής μνήμης, που θα δούμε στο επόμενο κεφάλαιο.



Σχήμα 2.1 *Χονδρική οργάνωση συστήματος* MIMD *κοινής μνήμης.*

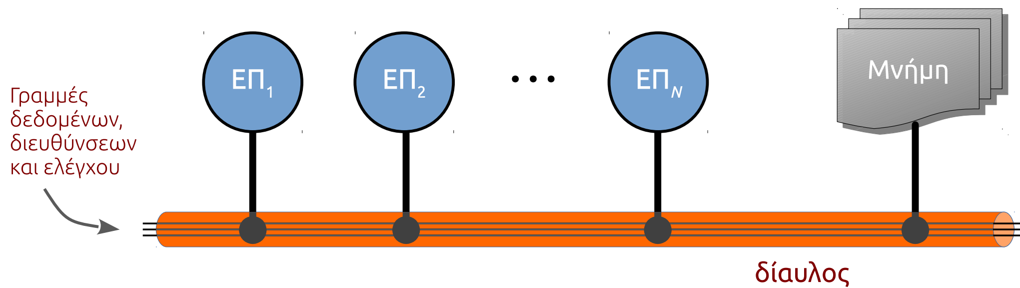
2.1 Βασική οργάνωση

Στην ενότητα αυτή, θα έχουμε την πρώτη επαφή μας με τους πολυεπεξεργαστές κοινόχρηστης μνήμης. Θα δούμε τη βασική δομή τους και κάποια από τα θέματα σχεδιασμού τους. Επίσης, θα γνωρίσουμε την πιο πετυχημένη, εμπορικά, κατηγορία συστημάτων MIMD κοινόχρηστης μνήμης, τους λεγόμενους συμμετρικούς πολυεπεξεργαστές.

Ένα σύστημα MIMD κοινόχρηστης μνήμης, όπως γνωρίζουμε από την Ενότητα 1.3, αποτελείται από ένα πλήθος N ανεξάρτητων επεξεργαστών, οι οποίοι προσπελαύνουν μία κοινόχρηστη κύρια μνήμη. Πολλές φορές η κύρια μνήμη έχει διασπαστεί σε μία συλλογή από μικρότερες μνήμες (έστω M στον αριθμό) , προκειμένου να επιτρέπονται ταυτόχρονες προσπελάσεις σε αυτές από διαφορετικούς επεξεργαστές. Η προσπέλαση γίνεται μέσω κάποιου δικτύου που ενώνει τους επεξεργαστές με τις μνήμες. Η βασική δομή ενός τέτοιου συστήματος φαίνεται στο Σχ. 2.1.

Η απλούστερη, οικονομικότερη και πιο συνηθισμένη μορφή που έχει το δίκτυο διασύνδεσης επεξεργαστών-μνημών είναι ένας δίαυλος (bus) και φαίνεται στο Σχ. 2.2. Στα εκατοντάδες καλώδια ενός τυπικού διαύλου, κυκλοφορούν οι διευθύνσεις, τα δεδομένα, σήματα χρονισμού και ελέγχου, που αφορούν την επικοινωνία των συσκευών. Στο δίαυλο είναι συνδεδεμένοι όλοι οι επεξεργαστές, καθώς και η κύρια μνήμη. Όλοι μπορούν να παρακολουθούν την κίνηση στον δίαυλο, αλλά μόνο ένα ζευγάρι συσκευών τη φορά μπορεί να τον χρησιμοποιεί για επικοινωνία.

Μία τέτοια οργάνωση είναι εντελώς φυσιολογική από τη στιγμή που σε οποιοδήποτε υπολογιστικό σύστημα θα υπάρχει κάποιος δίαυλος, τουλάχιστον για την επικοινωνία του επεξεργαστή με την κύρια μνήμη. Οι σύγχρονοι εμπορικοί επεξεργαστές, μάλιστα, έρχονται



Σχήμα 2.2 *Διασύνδεση με δίαυλο.*

εξοπλισμένοι με όλα εκείνα τα κυκλώματα που απαιτούνται για τη σύνδεσή τους σε κάποιον καθιερωμένο δίαυλο και δεν είναι, έτσι, καθόλου δύσκολο να ενσωματωθούν σε ένα τέτοιο σύστημα MIMD.

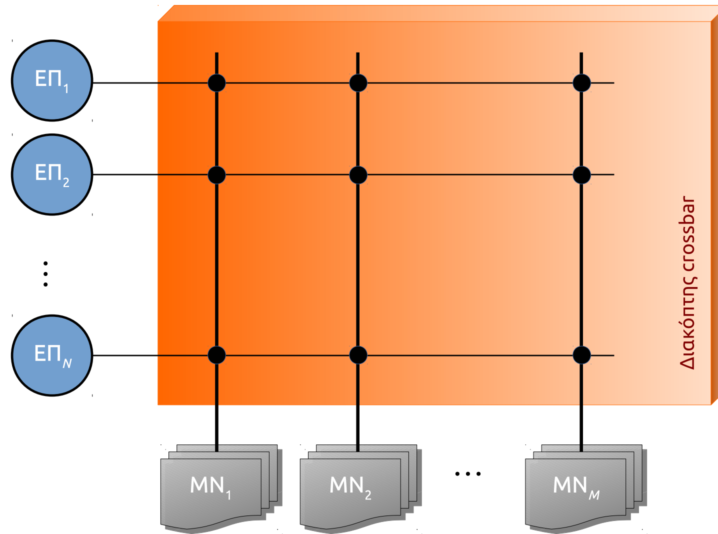
Η οργάνωση αυτή βοηθά πολύ και στο σχεδιασμό του λειτουργικού συστήματος της μηχανής. Εάν η εκτέλεση του λειτουργικού συστήματος δεν έχει ανατεθεί σε κάποιον από τους επεξεργαστές, αλλά μπορούν να συμμετέχουν όλοι ισότιμα (δεν υπάρχει δηλαδή σχέση αφέντη / σκλάβου μεταξύ των επεξεργαστών), το όλο σύστημα είναι γνωστό σαν *συμμετρικός πολυεπεξεργαστής* (symmetric multiprocessor, SMP).

Το βασικό πρόβλημα με τον δίαυλο είναι οι επιδόσεις του όταν αυξάνεται ο αριθμός των επεξεργαστών. Αυξάνοντας τους επεξεργαστές, αυξάνονται και οι αιτήσεις προς το δίαυλο για προσπελάσεις, ενώ η χωρητικότητα του διαύλου παραμένει σταθερή. Έτσι, γρήγορα φτάνουμε σε ένα σημείο κορεσμού όπου το σύστημα δεν μπορεί, πλέον, να ανταποκριθεί. Για τον λόγο αυτό, τα συστήματα που χρησιμοποιούν διασύνδεση με δίαυλο περιλαμβάνουν λίγους σχετικά επεξεργαστές (μονοψήφιο πλήθος συνήθως).

Προκειμένου να αντιμετωπιστεί αυτό το πρόβλημα, έχουν προταθεί και υλοποιηθεί κάποιες τροποποιήσεις, όπως χρήση παραπάνω του ενός διαύλου (π.χ. τέσσερις δίαυλοι στον Cray SuperServer 6400) ή ιεραρχία από διαύλους (π.χ. δύο επίπεδα διαύλων στους Encore Multimax και Ultramax). Μία άλλη λύση είναι η εγκατάλειψη του διαύλου για μία πιο ευέλικτη διασύνδεση επεξεργαστών-μνημών. Τέτοιου είδους διασυνδέσεις θα δούμε στην ενότητα που ακολουθεί.

2.2 Δίκτυα διακοπτών

Τα δίκτυα διακοπτών αντιπροσωπεύουν ένα άλλο είδος διασύνδεσης, το οποίο δεν στηρίζεται σε ένα κοινό μέσο που μπορούν να το παρακολουθούν όλοι οι επεξεργαστές. Αντίθετα, δημιουργούνται από κάθε επεξεργαστή σε κάθε μνήμη συγκεκριμένες διαδρομές που θα πρέπει να ακολουθηθούν, προκειμένου να μεταφερθούν τα δεδομένα. Στην ενότητα αυτή, θα δούμε τα σημαντικότερα από τα δίκτυα διακοπτών.



Σχήμα 2.3 *Διακόπτης crossbar.*

Το πιο γνωστό δίκτυο είναι ο διασταυρωτικός διακόπτης, τον οποίο θα δούμε στην Ενότητα 2.2.1. Στη συνέχεια (Ενότητα 2.2.2), θα κάνουμε μία εισαγωγή σε κάποια άλλα σημαντικά δίκτυα, τα λεγόμενα δίκτυα πολλαπλών επιπέδων (ή πολυεπίπεδα δίκτυα). Στην Ενότητα 2.2.3, θα δούμε μερικά ακόμα πολυεπίπεδα δίκτυα και θα ανακεφαλαιώσουμε τις κατηγορίες των δικτύων με διακόπτες.

Θα πρέπει να τονίσουμε εδώ ότι, αν και τα δίκτυα αυτά τα εξετάζουμε στα πλαίσια των επεξεργαστών κοινόχρηστης μνήμης, η χρήση τους είναι πολύ πιο ευρεία. Έχουν εφαρμοστεί τόσο σε συστήματα SIMD (π.χ. διακόπτης crossbar στον Burroughs BSP, πολυεπίπεδο δίκτυο στον STARAN) όσο και σε πολυεπεξεργαστές κατανεμημένης μνήμης για τη διασύνδεση μεταξύ των επεξεργαστών (π.χ. πολυεπίπεδα δίκτυα στους Thinking Machines CM-5, IBM SP-1 και SP-2, Meiko CS-2).

2.2.1 Διασταυρωτικός διακόπτης crossbar

Ο *διασταυρωτικός διακόπτης* (crossbar switch), ΔΔ, αποτελεί μία από τις πιο σημαντικές διασυνδέσεις. Ο λόγος είναι ότι πρόκειται, ίσως, για την λογικότερη (και αποτελεσματικότερη) οργάνωση ούτως ώστε να επιτρέπεται οποιαδήποτε σύνδεση μεταξύ των εμπλεκομένων μερών.

Στο Σχ. 2.3, φαίνεται η χονδρική δομή ενός ΔΔ που συνδέει N επεξεργαστές με M μνήμες (διακόπτης $N \times M$). Πρόκειται για ένα πλέγμα από $N \times M$ σημεία διασταύρωσης στα οποία καθορίζονται οι συνδέσεις, ανάλογα με τις αιτήσεις. Οριζοντίως, κάθε επεξεργαστής έχει πρόσβαση σε M σημεία διασταύρωσης για τη δημιουργία σύνδεσης με όποια κάθετη δίοδο οδηγεί στην επιθυμητή μνήμη. Κάθετα, για κάθε μνήμη υπάρχουν N σημεία διασταύρωσης για να μπορεί να συνδεθεί με οποιονδήποτε από τους N επεξεργαστές.

Προκειμένου, για παράδειγμα, ο επεξεργαστής i να προσπελάσει τη μνήμη j αρκεί να ενεργοποιηθεί η σύνδεση στο σημείο διασταύρωσης που ενώνει την i-οστή γραμμή με την j-οστή στήλη. Προσέξτε ότι μία μνήμη μπορεί να είναι συνδεδεμένη με έναν μόνο επεξεργαστή τη φορά, αφού η κάθε μνήμη διαθέτει μόνο μία γραμμή σύνδεσης προς τον διακόπτη. Είναι, όμως, δυνατόν να συνδέονται ταυτόχρονα διαφορετικοί επεξεργαστές με διαφορετικές μνήμες, εφόσον το κάθε συνδεδεμένο ζεύγος απασχολεί διαφορετική γραμμή και στήλη στον διακόπτη. Συγκρούσεις αιτήσεων δημιουργούνται μόνο όταν δύο διαφορετικοί επεξεργαστές προσπαθήσουν να προσπελάσουν μαζί την ίδια μνήμη.

Ο διασταυρωτικός διακόπτης έχει χρησιμοποιηθεί ευρέως και όχι μόνο σε πολυεπεξεργαστές κοινόχρηστης μνήμης. Είναι πρακτικός, όμως, όταν ο αριθμός των συνδεδεμένων στοιχείων δεν είναι πολύ μεγάλος. Αν υποθέσουμε ότι διαθέτουμε N επεξεργαστές και ισάριθμες μνήμες, τότε στον διακόπτη θα υπάρχουν N^2 σημεία διασταύρωσης, κάτι που κάνει το κόστος κατασκευής του υπερβολικά υψηλό (προσπαθήστε να σκεφτείτε τη λύση στο Πρόβλημα 2.1 και θυμηθείτε ότι, όπως και στον δίαυλο, προκειμένου να επικοινωνήσει ένας επεξεργαστής με μία μνήμη, απαιτούνται εκατοντάδες καλώδια). Ένας από τους μεγαλύτερους διακόπτες crossbar χρησιμοποιήθηκε στον Sun Enterprise 10000 (γνωστός ως «Starfire») και είναι μεγέθους 16×16, συνδέοντας 16 πλακέτες των 4 επεξεργαστών και 4 μνημών στην κάθε μία.

2.2.2 Δίκτυα πολλαπλών επιπέδων—το δίκτυο Δέλτα

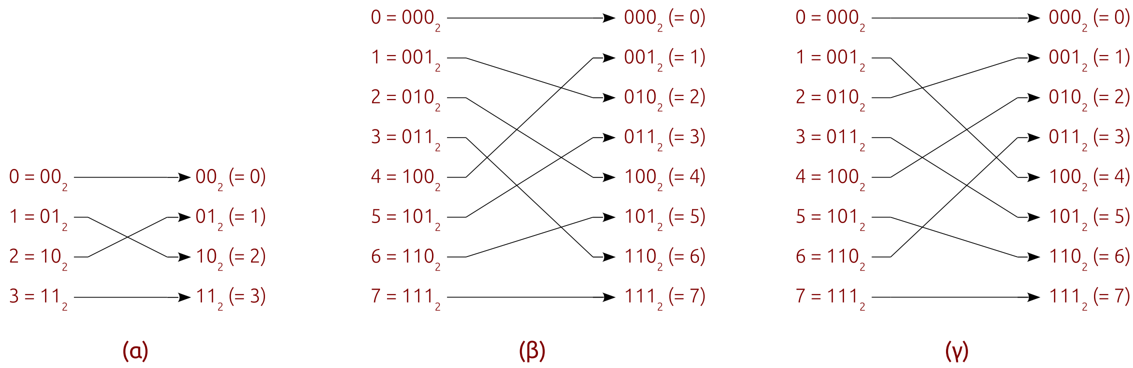
Λόγω της πολυπλοκότητας και του κόστους στην υλοποίηση ενός ΔΔ, όταν ο αριθμός των επεξεργαστών γίνεται μεγάλος, είναι απαραίτητη η χρήση δικτύων τα οποία αποτελούνται από μικρότερους ΔΔ συνδεδεμένους μεταξύ τους. Το αποτέλεσμα είναι τα λεγόμενα *δίκτυα πολλαπλών επιπέδων* (multistage networks) όπου μια σύνδεση επεξεργαστή-μνήμης περνά από αρκετά επίπεδα διακοπτών (αυξημένη καθυστέρηση), τα οποία όμως είναι πολύ πιο οικονομικά στην κατασκευή τους. Θα γνωρίσουμε τα δίκτυα αυτά μέσα από έναν συγκεκριμένο αντιπρόσωπό τους (το δίκτυο Δέλτα), αρκετά αναλυτικά. Περισσότερα δίκτυα θα δούμε στην επόμενη ενότητα.

Θα ασχοληθούμε με το λεγόμενο *συμμετρικό δίκτυο Δέλτα*, όπου ο αριθμός των εισόδων (επεξεργαστών), N, είναι ίσος με τον αριθμό των εξόδων (μνημών), M και ίσος με μία δύναμη του a, δηλαδή $N = M = a^k$ για κάποιο a και για κάποιο k. Για την περιγραφή της δομής του δικτύου αυτού είναι απαραίτητος ο ορισμός της πράξης *shuffle* (που σημαίνει ανακάτωμα).

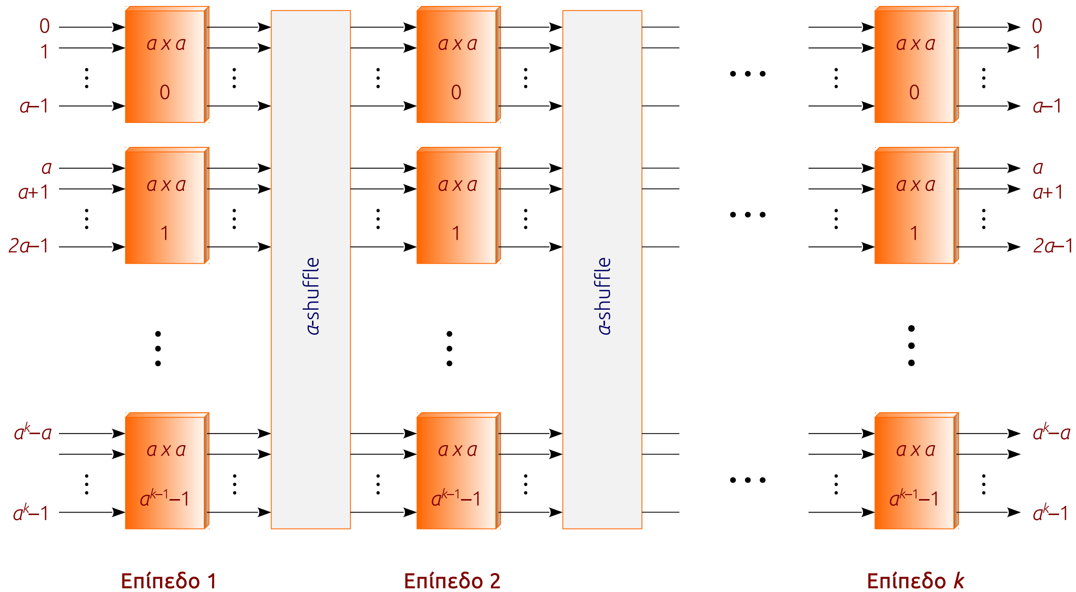
Έστω, λοιπόν, το σύνολο των k-ψήφιων αριθμών στο σύστημα αρίθμησης με βάση το a,

$$x = (x_k, x_{k-1}, \dots, x_1)_a ,$$

όπου, για κάθε ψηφίο x_i, i = 1, 2, ..., k, ισχύει $0 \le x_i \le a - 1$. Η πράξη a-shuffle (S_a) ορίζει



Σχήμα 2.4 *Η πράξη 2-shuffle (α) στους διψήφιους και (β) στους τριψήφιους δυαδικούς αριθμούς. Στο σχήμα (γ) εμφανίζεται η αντίστροφη 2-shuffle για τριψήφιους δυαδικούς αριθμούς.*



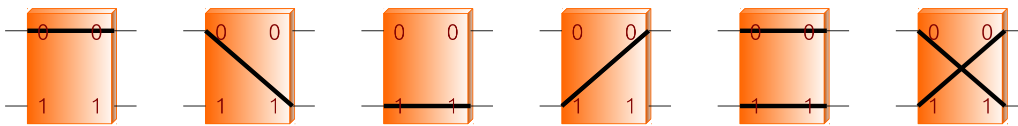
Σχήμα 2.5 *Η δομή του συμμετρικού δικτύου Δέλτα.*

ένα νέο αριθμό ως εξής:

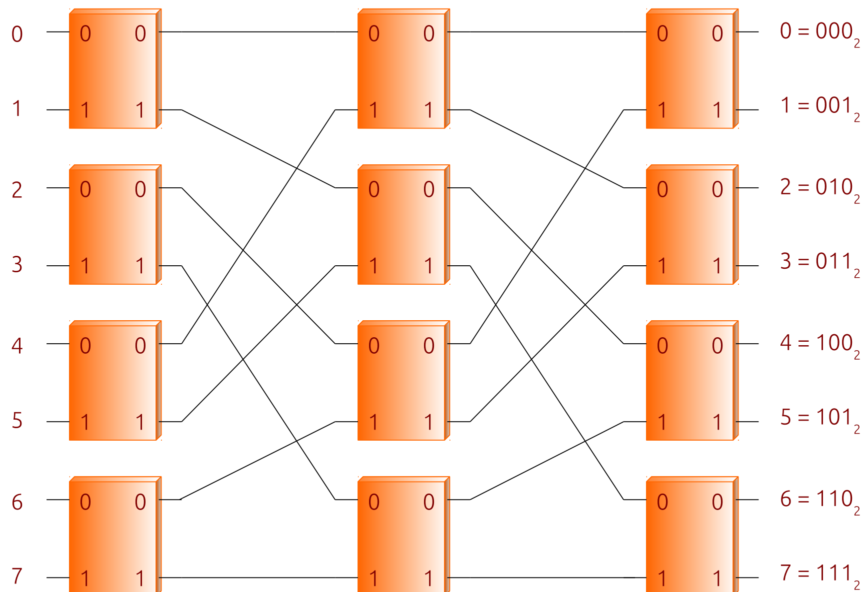
$$S_a(x) = (x_{k-1}, x_{k-2}, , \ldots, x_1, x_k)_a \,.$$

Το περισσότερο σημαντικό ψηφίο τοποθετείται στη λιγότερο σημαντική θέση, κάτι που είναι γνωστό και ως «περιστροφή» του αριθμού προς τα αριστερά. Αντίστοιχα, η «περιστροφή» του αριθμού προς τα δεξιά ορίζει την *αντίστροφη* (reverse) a-shuffle. Αν, για παράδειγμα, $a = 2$, τότε η πράξη S_2 δίνει την αντιστοιχία που φαίνεται στο Σχ. 2.4, για διψήφιους και τριψήφιους ($k = 2, 3$) δυαδικούς αριθμούς.

Το συμμετρικό δίκτυο Δέλτα $a^k \times a^k$ συνδέει a^k επεξεργαστές με a^k μνήμες και έχει τη μορφή που δίνεται στο Σχ. 2.5. Διαθέτει k επίπεδα διακοπτών. Σε κάθε επίπεδο υπάρχουν a^{k-1} διακόπτες με a εισόδους και a εξόδους ο καθένας, οι οποίοι είναι απλοί διακόπτες crossbar $a \times a$. Οι διακόπτες αυτοί μπορούν να συνδέσουν οποιαδήποτε είσοδό τους με



Σχήμα 2.6 *Οι 6 συνδέσεις εισόδων / εξόδων ενός crossbar 2×2.*



Σχήμα 2.7 *Το δίκτυο Δέλτα $2^3 \times 2^3$.*

οποιαδήποτε έξοδό τους και, μάλιστα, μπορούν να εξυπηρετήσουν ταυτόχρονα έως και a μη συγκρουόμενες αιτήσεις (οι δυνατές συνδέσεις για έναν διακόπτη 2×2 απεικονίζονται στο Σχ. 2.6). Οι a^k έξοδοι του ενός επιπέδου συνδέονται με τις a^k εισόδους του επομένου επιπέδου με βάση την αντιστοιχία της πράξης S_a. Στο Σχ. 2.7 φαίνεται το δίκτυο Δέλτα $2^3 \times 2^3$.

Από τη στιγμή που κάθε επίπεδο έχει a^{k-1} $(= N/a)$ διακόπτες και υπάρχουν k $(= \log_a N)$ επίπεδα, ένα δίκτυο Δέλτα αποτελείται συνολικά από $(N/a) \log_a N$ διακόπτες $a \times a$. Άρα, το κόστος κατασκευής τους είναι τάξης $O(N \log_a N)$, κάτι που αποτελεί σημαντική βελτίωση σε σχέση με το κόστος $O(N^2)$ που απαιτείται για έναν ΔΔ με ίδιο αριθμό συνδέσεων.

Φυσικά, η μείωση του κόστους συνεπάγεται και κάποιες ανάλογες μειώσεις στις επιδόσεις. Καταρχάς, είναι φανερό ότι κάθε αίτηση προς μία μνήμη πρέπει να περάσει από k στάδια διακοπτών, εισάγοντας έτσι σημαντικές καθυστερήσεις. Κατά δεύτερο λόγο, όπως θα δούμε παρακάτω, είναι δυνατόν να μην επιτρέπονται παράλληλες προσπελάσεις προς τις μνήμες κάτι το οποίο δεν συμβαίνει στον ΔΔ.

Θα πρέπει να τονίσουμε εδώ, ότι είναι δυνατή μία διασύνδεση N επεξεργαστών με M μνήμες, όπου $N \neq M$. Το δίκτυο που προκύπτει ονομάζεται *γενικευμένο δίκτυο Δέλτα*, βασίζεται στην γενικευμένη πράξη a-shuffle και δεν θα μας απασχολήσει εδώ.

Ας δούμε τώρα πώς μπορεί ένας επεξεργαστής ΕΠ$_x$ να συνδεθεί με μια μνήμη ΜΝ$_y$. Ο αλγόριθμος είναι εξαιρετικά απλός αλλά, κυρίως, είναι *κατανεμημένος*, δηλαδή ο κάθε διακόπτης, από τον οποίο περνάει η αίτηση, βρίσκει μόνος του την κατεύθυνση στην οποία πρέπει να την προωθήσει, προκειμένου να φτάσει στον προορισμό της. Ας υποθέσουμε ότι:

$$x = (x_k, x_{k-1}, \ldots, x_1)_a, \quad y = (y_k, y_{k-1}, \ldots, y_1)_a.$$

Αριθμώντας τις γραμμές στο Σχ. 2.5 από επάνω προς τα κάτω, ο επεξεργαστής x συνδέεται στη x-οστή είσοδο του πρώτου επιπέδου. Από τις a εισόδους του διακόπτη στον οποίο συνδέεται ο επεξεργαστής αυτός, η σύνδεση είναι στην x_1-οστή. Ο διακόπτης αυτός ρυθμίζεται (προγραμματίζεται) ώστε αυτή η είσοδος να συνδεθεί στην y_k-οστή έξοδό του. Δεν είναι δύσκολο να δούμε ότι η έξοδος αυτή είναι η έξοδος out_1 του πρώτου επιπέδου, όπου:

$$\mathrm{out}_1 = (x_k, x_{k-1}, \ldots, x_2, y_k)_a.$$

Λόγω της a-shuffle η έξοδος αυτή θα συνδεθεί στην είσοδο in_2 του δευτέρου επιπέδου:

$$\mathrm{in}_2 = (x_{k-1}, \ldots, x_2, y_k, x_k)_a.$$

Ο διακόπτης που έχει αυτή την είσοδο την συνδέει στη y_{k-1} έξοδό του η οποία είναι η out_2-οστή έξοδος στο δεύτερο επίπεδο, με:

$$\mathrm{out}_2 = (x_{k-1}, \ldots, x_2, y_k, y_{k-1})_a.$$

Η διαδικασία συνεχίζεται ως το επίπεδο k όπου η διαδρομή μας οδηγεί στην είσοδο:

$$\mathrm{in}_k = (y_k, y_{k-1}, \ldots, y_2, x_1)_a.$$

Ο διακόπτης που εμπλέκεται την συνδέει με την y_1-οστή έξοδό του και πλέον:

$$\mathrm{out}_k = (y_k, y_{k-1}, \ldots, y_2, y_1)_a,$$

η οποία συνδέεται στη μνήμη ΜΝ$_y$ που επιθυμούμε.

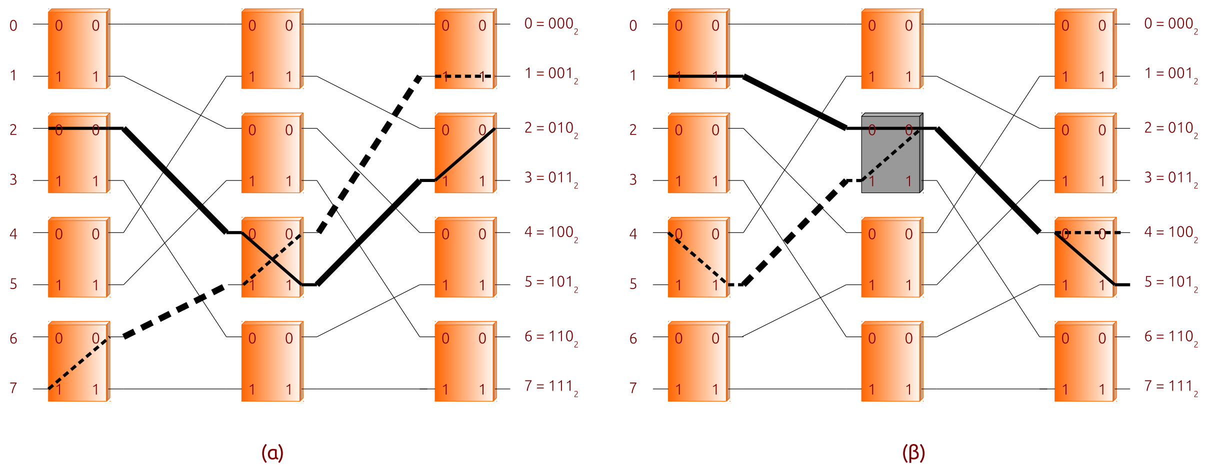
Η διαδικασία συνοψίζεται στον παρακάτω κανόνα για τον προγραμματισμό των διακοπτών, ώστε να σχηματιστεί μια διαδρομή από οποιονδήποτε επεξεργαστή σε οποιαδήποτε μνήμη ΜΝ$_y$, όπου $y = (y_k, y_{k-1}, \ldots, y_1)_a$:

Όταν ένας διακόπτης στο επίπεδο i λαμβάνει είσοδο για σύνδεση προς τη μνήμη ΜΝ$_y$, τη συνδέει με την y_{k-i+1} έξοδό του.

Με άλλα λόγια, όταν ένας διακόπτης λάβει αίτηση σε οποιαδήποτε είσοδό του για σύνδεση με τη μνήμη ΜΝ$_y$, η απόφαση για το αν η είσοδος αυτή θα συνδεθεί στην έξοδο 0, 1, ..., ή $a-1$, παίρνεται με βάση κάποιο ψηφίο του y.

Παράδειγμα 2.1

» *Διαδρομές στο δίκτυο Δέλτα*



Σχήμα 2.8 *Διαδρομές (α) και συγκρούσεις (β) στο δίκτυο Δέλτα* $2^3 \times 2^3$.

Θα ακολουθήσουμε βήμα προς βήμα τον παραπάνω κανόνα για να δούμε πώς σχηματίζεται η διαδρομή σύνδεσης μεταξύ του ζεύγους ΕΠ$_2$-ΜΝ$_2$ στο δίκτυο $2^3 \times 2^3$. Καταρχάς, $y = 2 = (y_3 y_2 y_1)_2 = 010_2$ είναι η δυαδική αναπαράσταση της μνήμης με την οποία θέλει να συνδεθεί ο ΕΠ$_2$. Η διαδρομή έχει ως εξής:

Ο επεξεργαστής συνδέεται στον δεύτερο (από πάνω, όπως κοιτάμε το Σχ. 2.7) διακόπτη του επιπέδου 1. Αυτός θα πρέπει να ελέγξει το $(3-1+1) = 3$ο bit του y το οποίο είναι ίσο με το 0. Επομένως, θα δημιουργήσει σύνδεση προς την έξοδο 0.

Στη συνέχεια, μέσω μίας shuffle η διαδρομή μας οδηγεί στον τρίτο διακόπτη του επιπέδου 2. Αυτός, ελέγχοντας το $(3-2+1) = 2$ο bit του y, το οποίο είναι ίσο με 1, δημιουργεί σύνδεση με την έξοδό του 1.

Τέλος, μετά από μία ακόμη shuffle καταλήγουμε στον 2ο διακόπτη του επιπέδου 3. Το $(3-3+1) = 1$ο bit του y είναι ίσο με 0 και η αίτηση οδηγείται στην έξοδο 0, η οποία είναι συνδεδεμένη με τη μνήμη ΜΝ$_2$. Η ολοκληρωμένη διαδρομή φαίνεται στο Σχ. 2.8(α).

Όπως είναι φανερό, ένας επεξεργαστής μέσω των διακοπτών και των shuffle μπορεί να συνδεθεί με οποιαδήποτε μνήμη, με κατάλληλο προγραμματισμό των διακοπτών. Η σύνδεση αυτή είναι *μοναδική*, δηλαδή, υπάρχει μία και μόνο μία διαδρομή μέσα στο δίκτυο που συνδέει ένα συγκεκριμένο ζεύγος επεξεργαστή-μνήμης. Επίσης, είναι δυνατές *ταυτόχρονες* συνδέσεις διαφορετικών επεξεργαστών με διαφορετικές μνήμες, όπως δείχνει το Σχ. 2.8(α) στο οποίο, εκτός από το ζεύγος ΕΠ$_2$-ΜΝ$_2$, επικοινωνούν και οι ΕΠ$_7$-ΜΝ$_1$.

Όμως, είναι πιθανόν να υπάρχουν και *συγκρούσεις* στο δίκτυο ακόμα και αν μόνο δύο διαφορετικοί επεξεργαστές χρειάζεται να συνδεθούν με δύο διαφορετικές μνήμες. Αυτό φαίνεται στο Σχ. 2.8(β), όπου ο επεξεργαστής 1 θέλει να συνδεθεί με τη μνήμη 5 και ο επε-

ξεργαστής 4 με τη μνήμη 4. Η σύγκρουση υπάρχει γιατί και οι δύο διαδρομές χρησιμοποιούν την ίδια έξοδο στο διακόπτη του δευτέρου επιπέδου. Αυτό φυσικά δεν συμβαίνει στη διασύνδεση με διακόπτη crossbar που είδαμε στην προηγούμενη ενότητα. Για το λόγο αυτό, οι επιδόσεις του δικτύου Δέλτα αναμένεται να είναι κατώτερες.

Η τελευταία παρατήρηση γεννά το εξής ερώτημα: ποιες συνδέσεις μπορούν να γίνουν ταυτόχρονα και ποιες όχι; Σε ένα συμμετρικό δίκτυο Δέλτα, όπου έχουμε τον ίδιο αριθμό εισόδων και εξόδων, με μια σύνδεση αντιστοιχίζουμε κάποιον αριθμό (είσοδο) από ένα σύνολο αριθμών σε ένα άλλο αριθμό (έξοδο) του ίδιου συνόλου. Με άλλα λόγια, οι συνδέσεις στο δίκτυο ουσιαστικά αντιπροσωπεύουν *μεταθέσεις* στο σύνολο των αριθμών $\{0, 1, \ldots, a^k - 1\}$. Επομένως, η παραπάνω ερώτηση μπορεί να διατυπωθεί ως: *ποιες μεταθέσεις επιτρέπονται και ποιες όχι;*

Η απάντηση είναι ότι το δίκτυο Δέλτα επιτρέπει μόνο ένα μικρό ποσοστό από όλες τις δυνατές μεταθέσεις (ενώ ο ΔΔ τις επιτρέπει όλες). Για τον λόγο αυτό, το δίκτυο Δέλτα ανήκει στην κατηγορία των λεγόμενων *εμποδιστικών δικτύων* (blocking networks). Τα δίκτυα που επιτρέπουν οποιαδήποτε μετάθεση λέγονται *μη-εμποδιστικά* (non-blocking) ή *παγκόσμια* (universal).

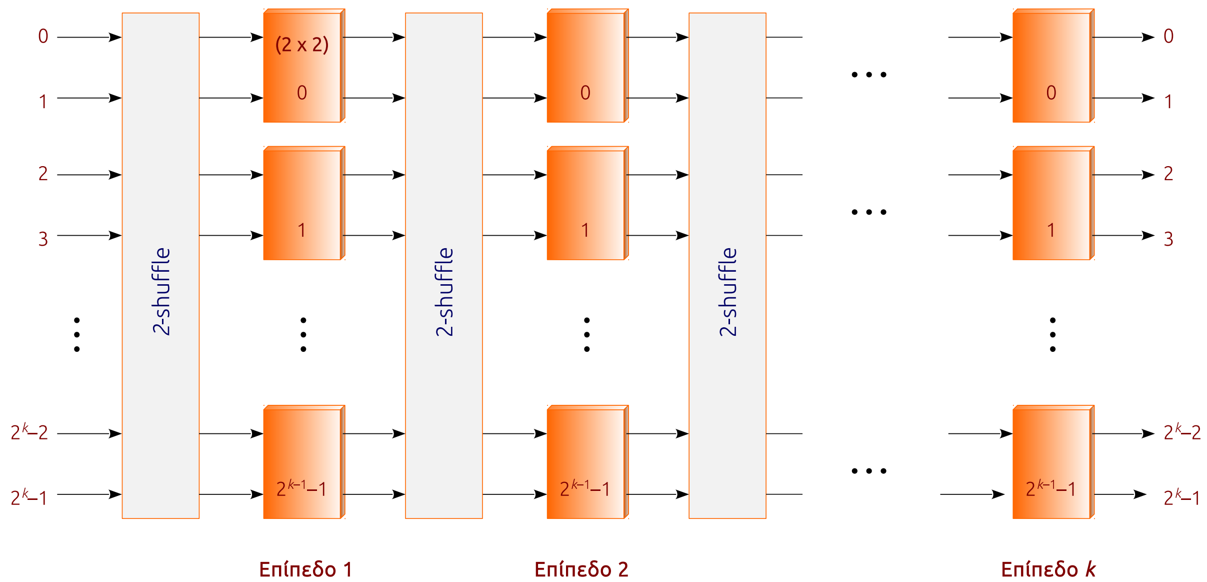
2.2.3 Άλλα δίκτυα

Εκτός από το δίκτυο Δέλτα, υπάρχουν πολλά άλλα δίκτυα πολλαπλών επιπέδων που έχουν μελετηθεί (και τα οποία έχουν τις ρίζες τους στους διακόπτες που σχεδιάστηκαν για τηλεφωνικά κέντρα). Τέτοια δίκτυα είναι τα Clos, baseline, butterfly Benes κλπ. Θα δούμε σύντομα μερικά από αυτά.

Το δίκτυο *Ωμέγα*, προτάθηκε ιστορικά πριν το δίκτυο Δέλτα, χρησιμοποιεί μόνο διακόπτες 2×2, και απεικονίζεται στο Σχ. 2.9. Με λίγη παρατήρηση μπορεί κανείς να δει ότι ουσιαστικά πρόκειται για ένα δίκτυο Δέλτα $2^k \times 2^k$ με μία επιπλέον σύνδεση shuffle πριν το πρώτο επίπεδο.

Ένα άλλο δίκτυο είναι το *baseline* το οποίο είναι επίσης $2^k \times 2^k$ και μπορεί να οριστεί με διάφορους τρόπους. Ένας από αυτούς είναι ο αναδρομικός, ο οποίος φαίνεται στο Σχ. 2.10. Στο ίδιο σχήμα δίνεται η κατασκευή του δικτύου baseline 4×4. Προσέξτε ότι η σύνδεση από το πρώτο επίπεδο διακοπτών βασίζεται στην *αντίστροφη* πράξη shuffle (απλά αλλάξτε τη φορά στα βέλη στο Σχ. 2.4). Όσο και να φαίνεται παράξενο, έχει αποδειχθεί ότι τα δίκτυα Δέλτα, Ωμέγα, baseline, καθώς και μερικά άλλα, είναι τοπολογικά ισοδύναμα μεταξύ τους.

Όλα τα δίκτυα, που είδαμε μέχρι στιγμής, είναι εμποδιστικά. Μη-εμποδιστικά δίκτυα είναι μόνο δύο: ο ΔΔ και το δίκτυο *Clos*, το οποίο ξεφεύγει από τους σκοπούς μας εδώ. Ένα άλλο δίκτυο, το οποίο αποτελεί μία ιδιαίτερη κατηγορία από μόνο του είναι το δίκτυο *Benes*. Το δίκτυο αυτό είναι *επαναδιατάξιμο* (rearrangeable), έτσι ώστε να επιτρέπει



Σχήμα 2.9 *Το δίκτυο Ωμέγα.*

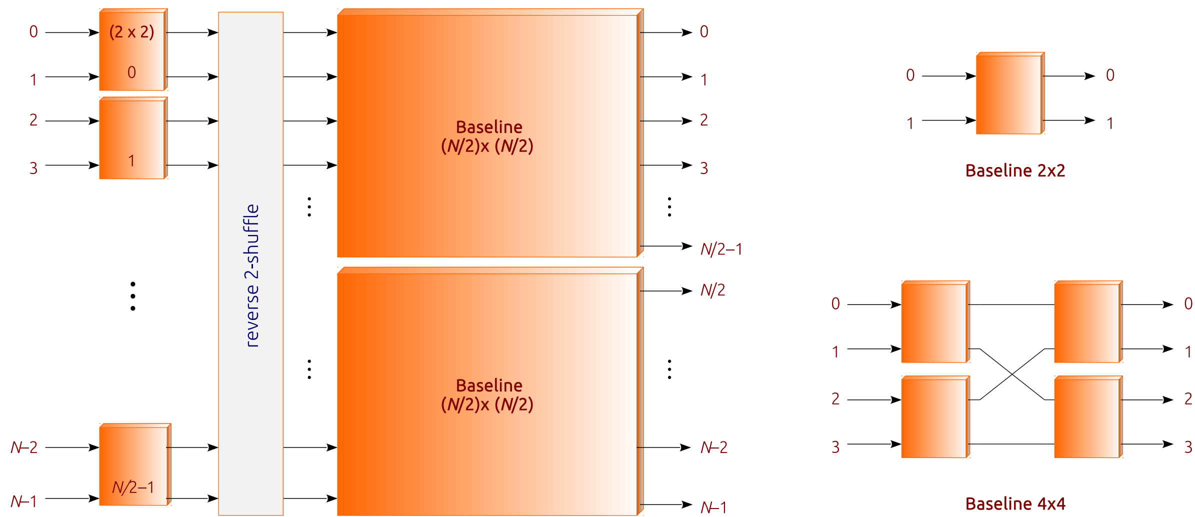
οποιεσδήποτε μεταθέσεις. Το δίκτυο αυτό δεν είναι μη-εμποδιστικό· απλά, διαθέτει παραπάνω από μία διαδρομές μεταξύ ενός επεξεργαστή και μίας μνήμης. Αν μία νέα σύνδεση δημιουργήσει σύγκρουση, το δίκτυο Benes μπορεί να προσαρμόσει τους διακόπτες και τις τρέχουσες συνδέσεις, ώστε να εξυπηρετήσει τη νέα σύνδεση με διαδρομή που δεν προκαλεί σύγκρουση.

Τέλος, σημειώνουμε ότι υπάρχουν και δίκτυα *επανακυκλοφορίας* (recirculating networks) τα οποία αποτελούνται από ένα μόνο επίπεδο διακοπτών, καθώς και συνδέσεις ανάδρασης από τις εξόδους στις εισόδους τους. Δεδομένα από έναν επεξεργαστή προς μία μνήμη θα πρέπει να κυκλοφορήσουν πολλές φορές μέσα-έξω στο δίκτυο, πριν φτάσουν στον προορισμό τους.

2.3 Ανάλυση επιδόσεων διασυνδέσεων

Θα δούμε τώρα, μέσα από ένα απλό αναλυτικό μοντέλο, τις επιδόσεις των διαφόρων διασυνδέσεων μεταξύ των επεξεργαστών και των κοινόχρηστων μνημών που γνωρίσαμε στις προηγούμενες ενότητες. Θα μπορέσουμε, έτσι, να συγκρίνουμε και ποσοτικά τις διασυνδέσεις με διαύλους, με διασταυρωτικούς διακόπτες και με δίκτυα πολλαπλών επιπέδων. Η ενότητα αυτή, πάντως, είναι εντελώς ανεξάρτητη από τις υπόλοιπες και δεν αποτελεί προαπαιτούμενη γνώση για τις επόμενες. Ο αναγνώστης που δεν ενδιαφέρεται για την πιθανοτική ανάλυση που ακολουθεί, μπορεί να την παραλείψει χωρίς πρόβλημα.

Για την απλοποίηση της μαθηματικής ανάλυσης που θα ακολουθήσει, χρειάζεται να κάνουμε κάποιες παραδοχές. Συγκεκριμένα, θα υποθέσουμε τα εξής:



Σχήμα 2.10 *Δίκτυα baseline.*

1. Υπάρχουν N επεξεργαστές και M μνήμες·
2. Το σύστημα είναι συγχρονισμένο με κοινό ρόλοι και λειτουργεί σε «κύκλους»· στην αρχή του κύκλου του ρολογιού, οι επεξεργαστές κάνουν τις αιτήσεις τους για προσπέλαση μνήμης και στο τέλος του κύκλου λαμβάνουν τα δεδομένα (αν η αίτηση γίνει δεκτή)·
3. Σε κάθε κύκλο, ένας επεξεργαστής ζητά δεδομένα με πιθανότητα p·
4. Οι προσπελάσεις που ζητούνται είναι *χωρικά ομοιόμορφες*, δηλαδή ένας επεξεργαστής προσπελαύνει οποιαδήποτε μνήμη με την ίδια πιθανότητα, $1/M$·
5. Οι προσπελάσεις είναι επίσης *χρονικά ανεξάρτητες*, δηλαδή, η επόμενη προσπέλαση που κάνει κάποιος επεξεργαστής είναι ανεξάρτητη από την προηγούμενη προσπέλασή του·
6. Όταν δύο ή παραπάνω επεξεργαστές προσπαθούν να προσπελάσουν την ίδια μνήμη, μόνο ένας «κερδίζει». Οι άλλοι συναγωνιζόμενοι «ξεχνούν» την αίτησή τους (δεν πρόκειται να περιμένουν να τελειώσει ο πρώτος επεξεργαστής). Η παραδοχή αυτή είναι φυσικά εξωπραγματική αλλά απλοποιεί το μοντέλο και διατηρεί την παραδοχή 4. Επίσης έχει παρατηρηθεί ότι δεν προκαλεί μεγάλες αποκλίσεις στους υπολογισμούς.

Με τις παραπάνω παραδοχές, μπορούμε να ξεκινήσουμε την ανάλυση, χωρίς να μας ενδιαφέρει (προς το παρόν) το είδος της διασύνδεσης. Η πιθανότητα να ζητηθεί κάποια μνήμη MN_k σε ένα κύκλο είναι ίση με:

$$q \quad = \quad P[\text{η } \text{MN}_k \text{ προσπελαύνεται από κάποιον επεξεργαστή}]$$

$$\begin{aligned}
&= 1 - P[\text{η ΜΝ}_k \text{ δεν προσπελαύνεται από κανέναν}] \\
&= 1 - P[\text{ο ΕΠ}_1 \text{ δεν προσπελαύνει την ΜΝ}_k] \times \cdots \\
&\qquad \times P[\text{ο ΕΠ}_N \text{ δεν προσπελαύνει την ΜΝ}_k] \\
&= 1 - P[\text{ένας ΕΠ δεν προσπελαύνει την ΜΝ}_k]^N.
\end{aligned}$$

Ένας συγκεκριμένος επεξεργαστής δεν προσπελαύνει τη μνήμη ΜΝ$_k$ αν:

- δεν ζητήσει καμία προσπέλαση (με πιθανότητα $1 - p$) ή αν
- ζητήσει προσπέλαση (με πιθανότητα p), η οποία είναι για οποιαδήποτε μνήμη εκτός της ΜΝ$_k$ (με πιθανότητα $1 - 1/M$),

δηλαδή,

$$P[\text{ένας ΕΠ δεν προσπελαύνει την ΜΝ}_k] = (1-p) + p\left(1 - \frac{1}{M}\right) = 1 - \frac{p}{M},$$

και επομένως, η πιθανότητα να ζητηθεί μια συγκεκριμένη μνήμη ΜΝ$_k$ είναι, για κάθε k, ίση με:

$$q = 1 - \left(1 - \frac{p}{M}\right)^N \tag{2.1}$$

Αφού μία μνήμη ζητείται με πιθανότητα q, θα ζητηθούν συνολικά i μνήμες με πιθανότητα:

$$f(i) = \binom{M}{i} q^i (1-q)^{M-i}.$$

Αυτό συμβαίνει διότι υπάρχουν $\binom{M}{i}$ τρόποι να επιλεχθούν i μνήμες σε σύνολο M μνημών, και για κάθε έναν από αυτούς ακριβώς i μνήμες προσπελαύνονται (q^i), ενώ οι υπόλοιπες $M - i$ όχι ($(1-q)^{M-i}$).

2.3.1 Επιδόσεις διαύλων

Θα δούμε τώρα τι συμβαίνει στην περίπτωση που διαθέτουμε ένα σύστημα με διασύνδεση διαύλων. Συγκεκριμένα, θα υποθέσουμε ότι υπάρχουν $B \geq 1$ ανεξάρτητοι δίαυλοι, επάνω στους οποίους είναι συνδεδεμένοι όλοι οι επεξεργαστές και όλες οι μνήμες.

Οι επιδόσεις του συστήματος καθορίζονται από το πόσες μνήμες είναι απασχολημένες (κατά μέσο όρο) σε κάθε κύκλο, το οποίο είναι το ίδιο με τον αριθμό των διαύλων που είναι σε χρήση σε κάθε κύκλο. Μπορεί να έχουμε B διαύλους αλλά, λόγω των συγκρούσεων στις αιτήσεις των επεξεργαστών, δεν είναι δυνατόν να χρησιμοποιούνται πάντα και οι B δίαυλοι. Επομένως, ο ουσιαστικός (μέσος) αριθμός απασχολημένων διαύλων θα είναι:

$$\begin{aligned}
\text{BW}_b(N, M, B) &= \sum_{i=1}^{M} f(i) \times (\#\ \text{χρησιμοπ. διαύλων όταν υπάρχουν } i \text{ αιτήσεις}) \\
&= \sum_{i=1}^{B} f(i) \times i + \sum_{i=B+1}^{M} f(i) \times B,
\end{aligned} \tag{2.2}$$

αφού δεν μπορούν να γίνουν παραπάνω από B προσπελάσεις (υποθέσαμε φυσικά ότι $B < M$). Στην (2.2) όλα είναι γνωστά και δίνουν το BW για οποιοδήποτε συνδυασμό των N, M και B. Ο μέσος αριθμός απασχολημένων διαύλων / μνημών (BW) ονομάζεται και *εύρος ζώνης* της διασύνδεσης (bandwidth).

Για να δούμε ποιοτικά πώς ο αριθμός των διαύλων B επηρεάζει τις επιδόσεις του συστήματος, μπορούμε να υπολογίσουμε εύκολα από την (2.2) τα όρια μέσα στα οποία κινείται το $\mathrm{BW}_\delta(N, M, B)$. Το πρώτο όριο δίνεται από τη σχέση:

$$\mathrm{BW}_b(N, M, B) \leq B,$$

αφού υπάρχουν μόνο B δίαυλοι στο σύστημα. Από την άλλη, στο δεύτερο άθροισμα της (2.2), το B είναι πάντα μικρότερο του i, άρα:

$$\mathrm{BW}_b(N, M, B) \leq \sum_{i=1}^{B} f(i) \times i + \sum_{i=B+1}^{M} f(i) \times i = \sum_{i=1}^{M} i f(i).$$

Προσέξτε ότι το $\sum_{i=1}^{M} i f(i)$ είναι ό μέσος αριθμός μνημών που προσπελαύνονται, και θα πρέπει να είναι ίσο με Mq, αφού υπάρχουν M μνήμες και κάθε μία έχει πιθανότητα q να προσπελαστεί, άρα,

$$\mathrm{BW}_b(N, M, B) \leq Mq.$$

Οι επιδόσεις περιορίζονται (α) από τον αριθμό των διαύλων (και επομένως, αυξάνοντας τον αριθμό τους θα έχουμε και αύξηση των επιδόσεων), και (β) από την ποσότητα Mq. Από κάποιο σημείο και έπειτα (όταν δηλαδή $B > Mq$), ο δεύτερος περιορισμός υπερισχύει· όσο και να αυξηθεί το B, δεν υπάρχει ουσιαστικά βελτίωση στην απόδοση. Για την πλέον συνήθη περίπτωση του ενός διαύλου (όπου $B = N = 1$), έχουμε:

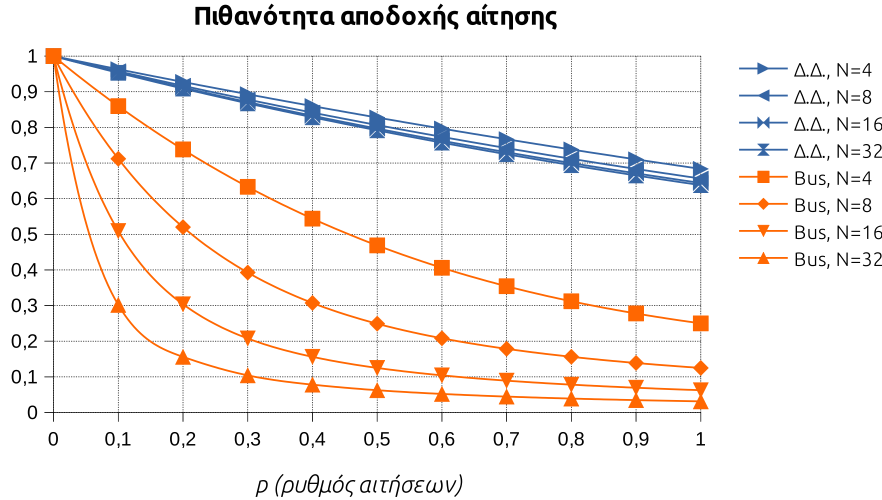
$$\mathrm{BW}_b(N, 1, 1) = q = 1 - (1 - p)^N. \tag{2.3}$$

2.3.2 Επιδόσεις διασταυρωτικών διακοπτών

Η ανάλυση των επιδόσεων ενός ΔΔ csrossbar είναι πολύ παρόμοια με αυτή της προηγούμενης παραγράφου. Για να υπολογίσουμε το μέσο αριθμό μνημών που χρησιμοποιούνται σε κάθε κύκλο ρολογιού (bandwidth), χρησιμοποιούμε τη σχέση (2.1) η οποία μας δίνει την πιθανότητα να ζητηθεί μια συγκεκριμένη μνήμη. Στη συγκεκριμένη διασύνδεση, όμως, δεν έχουμε κάποιο περιορισμό από διαύλους και όλες οι προσπελάσεις των (διαφορετικών) μνημών θα ολοκληρωθούν.

Αφού υπάρχουν M μνήμες, και η κάθε μία προσπελαύνεται με πιθανότητα q, κατά μέσο όρο θα υπάρχουν:

$$\mathrm{BW}_c(N, M) = Mq = M - M\left(1 - \frac{p}{M}\right)^N \tag{2.4}$$



Σχήμα 2.11 *Επιδόσεις διαύλων και διασταυρωτικών διακοπτών για* $N = 4, 8, 16, 32$ *επεξεργαστές.*

μνήμες που απασχολούνται σε κάθε κύκλο. Πρέπει να είναι φανερό ότι ανεξαρτήτως του αριθμού των διαύλων,

$$\mathrm{BW}_c(N, M) \geq \mathrm{BW}_b(N, M, B).$$

Εκτός από το BW, ένα άλλο ενδεικτικό μέτρο των επιδόσεων του συστήματος είναι η πιθανότητα, A_c, που έχει κάποια αίτηση για προσπέλαση να γίνει δεκτή· όσο μεγαλύτερη είναι αυτή η πιθανότητα, τόσο πιο πολλές προσπελάσεις μπορούν να γίνουν ταυτόχρονα χωρίς συγκρούσεις και επομένως, τόσο πιο πολλές μνήμες θα χρησιμοποιούνται.

Κατά μέσο όρο, σε ένα κύκλο του συστήματος, γίνονται pN αιτήσεις για προσπέλαση από τους N επεξεργαστές, ενώ το σύστημα υποστηρίζει $\mathrm{BW}_c(N, M)$ προσπελάσεις. Άρα:

$$A_c = \frac{\mathrm{BW}_c(N, M)}{pN} = \frac{M}{pN} - \frac{M}{pN}\left(1 - \frac{p}{M}\right)^N.$$

Αν το πλήθος των επεξεργαστών και των μνημών είναι ίδιο ($N = M$), η πιθανότητα αποδοχής μίας αίτησης είναι ίση με:

$$A_c = \frac{1 - (1 - p/N)^N}{p}.$$

Αντίστοιχα, για έναν δίαυλο θα έχουμε, από την (2.3):

$$A_b = \frac{\mathrm{BW}_b(N, 1, 1)}{pN} = \frac{1 - (1 - p)^N}{pN}.$$

Στο Σχ. 2.11, δίνεται η γραφική αναπαράσταση των τελευταίων σχέσεων για την περίπτωση που το σύστημα διαθέτει $N = 4, 8, 16, 32$ επεξεργαστές. Η πιθανότητα να ολοκληρωθεί μία αίτηση όπως είναι αναμενόμενο μειώνεται, καθώς αυξάνει ο ρυθμός των

αιτήσεων από τους επεξεργαστές. Αυτό που φαίνεται καθαρά είναι ότι όσο περισσότεροι είναι οι επεξεργαστές τόσο χειρότερη είναι η επίδοση του διαύλου. Είναι κάτι που έχει επισημανθεί πολλές φορές μέχρι τώρα, αλλά πλέον, αποδεικνύεται με ιδιαίτερα εύγλωττο τρόπο. Ο διασταυρωτικός διακόπτης διατηρεί τις επιδόσεις του σε πολύ υψηλό σημείο.

2.3.3 Επιδόσεις πολυεπίπεδων δικτύων

Ένα δίκτυο Δέλτα αποτελείται από διακόπτες crossbar $a \times a$. Κάνοντας τις ίδιες παραδοχές με την Ενότητα 2.3, μπορούμε να εφαρμόσουμε τον τύπο (2.4) σε κάθε ένα από τους ΔΔ του δικτύου Δέλτα. Για να γίνει αυτό όμως, θα πρέπει να αποδείξουμε ότι η ομοιομορφία των προσπελάσεων ισχύει σε κάθε ΔΔ, δηλαδή ότι σε καθέναν από αυτούς, κάθε είσοδός του έχει την ίδια πιθανότητα να απασχολείται όπως και κάθε έξοδός του, αντίστοιχα.

Αν ο προορισμός είναι η μνήμη $y = (y_k, y_{k-1}, \ldots, y_1)_a$, το $(k - i + 1)$-οστό ψηφίο, και μόνο αυτό, ρυθμίζει σε ποια έξοδο ενός διακόπτη στο επίπεδο i θα γίνει η σύνδεση. Από τη στιγμή που οι προορισμοί (μνήμες) είναι ομοιόμορφα κατανεμημένοι, το ίδιο θα συμβαίνει και με κάθε ψηφίο των προορισμών. Άρα, κάποια είσοδος σε ένα διακόπτη στο επίπεδο i (για κάθε i), θα συνδέεται με την ίδια πιθανότητα σε οποιαδήποτε έξοδό του. Επομένως, οι έξοδοι κάθε επιπέδου i είναι ομοιόμορφα κατανεμημένες και, κατά συνέπεια, το ίδιο συμβαίνει και για τις εισόδους του επόμενου επιπέδου (οι οποίες είναι οι έξοδοι του προηγούμενου επιπέδου).

Με βάση τα παραπάνω, και χρησιμοποιώντας πλέον τον τύπο (2.4), αν σε κάποιο επίπεδο κάθε είσοδος ενός διακόπτη ζητά σύνδεση με πιθανότητα (ρυθμό) p_{in}, τότε ο μέσος αριθμός συνδέσεων BW που θα γίνουν προς τις εξόδους του θα είναι:

$$\text{BW} = a - a\left(1 - \frac{p_{\text{in}}}{a}\right)^a.$$

Αφού έχουμε a εξόδους, η κάθε μία θα έχει πιθανότητα:

$$p_{\text{out}} = \frac{\text{BW}}{a} = 1 - \left(1 - \frac{p_{\text{in}}}{a}\right)^a$$

να χρησιμοποιηθεί. Οι έξοδοι ενός επιπέδου i, όμως, είναι οι είσοδοι του επόμενου επιπέδου $i + 1$. Άρα, αν στο επίπεδο i, ο ρυθμός εξόδου (δηλ. η πιθανότητα να χρησιμοποιείται μια έξοδος) είναι p_i, θα έχουμε:

$$p_i = 1 - \left(1 - \frac{p_{i-1}}{a}\right)^a$$

Καταλήγουμε, επομένως, στον αναδρομικό τύπο:

$$\begin{aligned} p_0 &= p \\ p_i &= 1 - \left(1 - \frac{p_{i-1}}{a}\right)^a \qquad i = 1, 2, ..., k, \end{aligned}$$

όπου p είναι ό ρυθμός με τον οποίο κάθε επεξεργαστής αιτείται προσπελάσεις. Κάθε μία από τις a^k εξόδους του δικτύου, οι οποίες είναι έξοδοι του επιπέδου k, θα έχει πιθανότητα χρήσης p_k, οπότε:

$$\mathrm{BW}_\Delta(a^k \times a^k) = p_k a^k$$

και η πιθανότητα ότι κάποια αίτηση θα γίνει δεκτή, είναι ίση με:

$$A_\Delta = \frac{p_k a^k}{p a^k} = \frac{p_k}{p}.$$

2.4 Κρυφές μνήμες και το πρόβλημα της συνοχής

Προκειμένου να επιταχυνθούν οι προσπελάσεις στα δεδομένα στις σειριακές αρχιτεκτονικές, είναι πάντα παρούσα η κρυφή μνήμη (cache), όπως είναι γνωστό. Πιο συγκεκριμένα, πρόκειται για μία ιεραρχία από 2-3 επίπεδα μικρών σε μέγεθος αλλά ταχύτατων μνημών που έχουν ως στόχο τη διατήρηση και γρήγορη απόδοση των δεδομένων που χρησιμοποιεί πιο συχνά ο επεξεργαστής. Η ανάγκη αυτή είναι ακόμα μεγαλύτερη στους πολυεπεξεργαστές κοινόχρηστης μνήμης. Σε αυτή την ενότητα, θα εισάγουμε την κρυφή μνήμη στους επεξεργαστές του συστήματός μας αλλά, ταυτόχρονα, θα ανακαλύψουμε ένα νέο πρόβλημα που δημιουργείται με τη χρήση της.

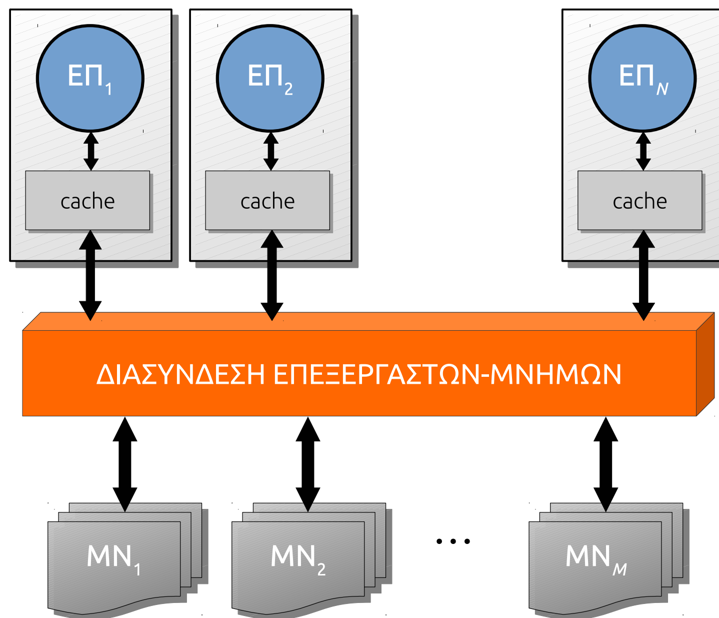
Μία βασική απαίτηση, που πάντα καλείται ο σχεδιαστής να ικανοποιήσει, είναι η γρήγορη προσπέλαση της μνήμης. Η απαίτηση είναι ακόμα πιο έντονη στα συστήματα που εξετάζουμε σε αυτό το κεφάλαιο και αυτό διότι, εκτός από την διαφορά ταχύτητας μεταξύ επεξεργαστών και μνημών, υπάρχουν δύο νέες πηγές επιβράδυνσης:

(α) ο ανταγωνισμός των επεξεργαστών για προσπέλαση της ίδιας μνήμης,

(β) οι καθυστερήσεις λόγω του δικτύου διασύνδεσης.

Για το (α) φανταστείτε ένα παράλληλο πρόγραμμα το οποίο αποτελείται από μία συλλογή διεργασιών με μία κοινή μεταβλητή μεταξύ τους. Κάθε επεξεργαστής αναλαμβάνει την εκτέλεση διαφορετικής διεργασίας, όλοι όμως θα προσπαθούν να προσπελάσουν την ίδια μνήμη, όταν χρειαστεί να διαβάσουν ή να τροποποιήσουν την κοινή μεταβλητή. Από τη στιγμή που η μνήμη θα μπορεί να εξυπηρετήσει έναν επεξεργαστή την φορά, βλέπουμε ότι τυχόν ταυτόχρονες προσπελάσεις τελικά θα σειριοποιηθούν, καθυστερώντας κάποιους επεξεργαστές.

Ακόμα και να μην υπάρχει νταγωνισμός μεταξύ των επεξεργαστών για τα κοινά δεδομένα, μπορεί να υπάρχει καθυστέρηση που οφείλεται στο δίκτυο διασύνδεσης (β). Η καθυστέρηση στα πολυεπίπεδα δίκτυα είναι κάτι παραπάνω από προφανής, αφού κάθε αίτηση χρειάζεται να διανύσει μία μη αμελητέα διαδρομή από διακόπτες. Στους διαύλους



Σχήμα 2.12 *Πολυεπεξεργαστής κοινόχρηστης μνήμης με κρυφές μνήμες.*

από την άλλη μεριά, υπάρχουν οι καθυστερήσεις λόγω ανταγωνισμού για την κατάληψη του κοινού μέσου, ακόμα και αν οι αιτήσεις είναι για διαφορετικές μνήμες.

Η λύση στα προβλήματα των αυξημένων καθυστερήσεων είναι η χρήση κρυφής μνήμης σε κάθε επεξεργαστή, όπως απεικονίζεται στο Σχ. 2.12. Η βελτίωση είναι, πραγματικά, πολύ σημαντική. Με μία σχετικά μικρή σε μέγεθος κρυφή μνήμη, είναι δυνατόν να φυλάσσονται με μεγάλη πιθανότητα τα δεδομένα που χρειάζεται πιο συχνά ο κάθε επεξεργαστής. Έτσι, τις περισσότερες φορές θα αποφεύγεται εντελώς το δίκτυο διασύνδεσης, με όλα τα ευεργετικά αποτελέσματα που αυτό συνεπάγεται.

Ένα ακόμη θετικό σημείο, που δεν φαίνεται με την πρώτη ματιά, είναι ότι ο απλός δίαυλος μπορεί, πλέον, να «σηκώσει» παραπάνω επεξεργαστές, αφού κατά μέσο όρο θα μειωθεί η κίνησή του με τη χρήση των κρυφών μνημών.

Δυστυχώς, όμως, η εισαγωγή της κρυφής μνήμης δεν επιφέρει μόνο βελτιώσεις. Αντίθετα, δημιουργεί ένα πολύ σημαντικό πρόβλημα, γνωστό ως *πρόβλημα συνοχής της κρυφής μνήμης* (cache coherency problem), το οποίο γίνεται εύκολα κατανοητό μέσα από το παράδειγμα που ακολουθεί.

Φανταστείτε δύο επεξεργαστές Α και Β και ένα δεδομένο Χ. Αν οι επεξεργαστές εκτελούν δύο διεργασίες που έχουν το Χ ως κοινή μεταβλητή, τότε αργά ή γρήγορα αντίγραφα του Χ θα βρεθούν και στις δύο ιδιωτικές, κρυφές μνήμες. Το πρόβλημα της συνοχής εμφανίζεται όταν κάποιος επεξεργαστής προσπαθήσει να τροποποιήσει τη μεταβλητή Χ. Εάν οι κρυφές μνήμες χρησιμοποιούν πολιτική υστεροεγγραφής (write-back), τότε η τροποποίηση του Χ, π.χ. στην κρυφή μνήμη του Α δεν θα γίνει γνωστή ούτε στην κύρια μνήμη ούτε στον επεξεργαστή Β. Αν και ο Β τροποποιήσει το δικό του αντίγραφο, το αποτέλε-

σμα θα είναι να υπάρχουν στο σύστημα τρεις διαφορετικές τιμές του X, χωρίς να γνωρίζει κανείς ποια είναι η σωστή!

Η συνοχή των κρυφών μνημών είναι από τα σημαντικότερα προβλήματα των πολυεπεξεργαστών κοινόχρηστης μνήμης και απαιτεί αποδοτικές λύσεις. Μια απλή ιδέα, η οποία λύνει το πρόβλημα, είναι να αποφευχθεί η εμφάνισή του σε επίπεδο *λογισμικού*. Συγκεκριμένα, απαιτείται κατάλληλος μεταφραστής (compiler) ο οποίος ξεχωρίζει και σημειώνει τα δεδομένα που χρησιμοποιούν οι διεργασίες, ως κοινόχρηστα ή ως ιδιωτικά. Κάθε ιδιωτικό δεδομένο το χρησιμοποιεί μόνο μία διεργασία και άρα, θα αντιγραφεί το πολύ σε μία κρυφή μνήμη. Τα κοινά δεδομένα, όμως, που προκαλούν το πρόβλημα της συνοχής, *δεν επιτρέπεται να αντιγραφούν σε καμία κρυφή μνήμη*. Με αυτόν τον τρόπο, δεν υπάρχει ποτέ περίπτωση να εμφανιστεί το πρόβλημα της συνοχής. Η λύση φαίνεται σχετικά απλή, απαιτεί όμως κατάλληλο μεταφραστή που να τοποθετεί ετικέτες στα δεδομένα ή πολύ προσεκτικό προγραμματισμό, στην περίπτωση που δεν υπάρχει τέτοιος μεταφραστής. Εκτός από αυτό, η καθυστέρηση κατά την προσπέλαση των κοινών δεδομένων (αφού θα πρέπει να γίνεται πάντα μέσω της κύριας μνήμης) δεν είναι γενικά αποδεκτή.

Η συνήθης πρακτική, όμως, είναι η αντιμετώπιση του προβλήματος σε επίπεδο υλικού. Σε κάποια παλαιότερα συστήματα έγινε προσπάθεια, όπως στη λύση σε επίπεδο λογισμικού, να αποτραπεί η εμφάνιση του προβλήματος. Για παράδειγμα, στο σύστημα Alliant FX-8 της δεκαετίας του 1980, υπήρχαν διαμοιραζόμενες κρυφές μνήμες στις οποίες είχαν πρόσβαση όλοι οι επεξεργαστές και έτσι, δεν υπήρχε ανάγκη για πολλαπλά αντίγραφα δεδομένων. Σήμερα, χρησιμοποιούνται σχεδόν αποκλειστικά αρχιτεκτονικές λύσεις, οι οποίες *επιτρέπουν* την εμφάνιση του προβλήματος και το επιλύουν κατά τη διάρκεια της εκτέλεσης των προγραμμάτων, μέσω «συνεννοήσεων» μεταξύ των κρυφών μνημών και της κύριας μνήμης. Ουσιαστικά, κάθε προσπέλαση υπόκειται διαφανώς σε κάποιον έλεγχο για να είναι σίγουρο ότι τα δεδομένα έχουν την πιο πρόσφατη τιμή.

Οι αρχιτεκτονικές αυτές λύσεις κατατάσσονται σε δύο κατηγορίες και θα τις δούμε στις επόμενες δύο ενότητες. Η πρώτη κατηγορία περιλαμβάνει τα λεγόμενα *πρωτόκολλα παρακολούθησης* (snooping protocols) και απαιτεί όλοι οι επεξεργαστές να παρακολουθούν συνεχώς τις αιτήσεις και την κίνηση των δεδομένων από / προς τις μνήμες. Επομένως, είναι εφαρμόσιμη μόνο αν το δίκτυο διασύνδεσης είναι δίαυλος (αφού όλοι οι επεξεργαστές είναι συνδεδεμένοι στο κοινό μέσο και μπορούν να το παρακολουθούν).

Αντίθετα, τα *πρωτόκολλα καταλόγων* (directory-based protocols) δεν απαιτούν κάποιο κοινόχρηστο μέσο παρακολούθησης. Όποιος επεξεργαστής θελήσει να διαβάσει ή να τροποποιήσει ένα δεδομένο, επικοινωνεί με κάποια μνήμη η οποία περιέχει καταλόγους με το ποιος επεξεργαστής έχει ποιο δεδομένο στην κρυφή μνήμη του. Η μνήμη, στη συνέχεια, ειδοποιεί όλους τους εμπλεκόμενους επεξεργαστές (και μόνο αυτούς) για τις επικείμενες ενέργειες.

2.5 Πρωτόκολλα πρακολούθησης

Στην ενότητα αυτή, θα γνωρίσουμε τα πρωτόκολλα παρακολούθησης για το πρόβλημα της συνοχής της κρυφής μνήμης. Όπως ήδη είπαμε, μία τέτοιου είδους λύση έχει εφαρμογή μόνο όταν το δίκτυο διασύνδεσης είναι δίαυλος.

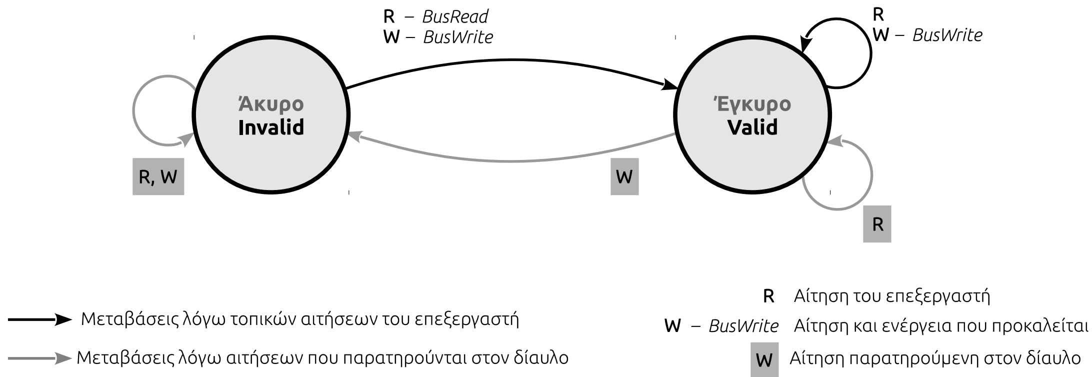
Τα πρωτόκολλα παρακολούθησης χωρίζονται σε δύο κατηγορίες ανάλογα με την πολιτική πληροφόρησης των υπολοίπων κρυφών μνημών, όταν τροποποιηθούν δεδομένα σε κάποια κρυφή μνήμη. Συγκεκριμένα, μπορεί να είναι *εγγραφής-ενημέρωσης* (write-update) ή *εγγραφής-ακύρωσης* (write-invalidate).

Στα πρωτόκολλα εγγραφής-ενημέρωσης, όταν ένα δεδομένο τροποποιηθεί στην κρυφή μνήμη ενός επεξεργαστή, η νέα τιμή μεταδίδεται σε όλους τους επεξεργαστές. Ως αποτέλεσμα, όσες κρυφές μνήμες έχουν το δεδομένο, το ενημερώνουν αμέσως με τη νέα του τιμή. Αντίθετα, στα πρωτόκολλα εγγραφής-ακύρωσης, η νέα, τροποποιημένη τιμή του δεδομένου δεν μεταδίδεται στις υπόλοιπες κρυφές μνήμες. Απλά, όποια διαθέτει το δεδομένο, το ακυρώνει. Όταν αργότερα το χρειαστεί ο αντίστοιχος επεξεργαστής, θα προκληθεί αστοχία (miss) και το δεδομένο θα ζητηθεί από την κύρια μνήμη ή από κάποια άλλη κρυφή μνήμη.

Η επιλογή ανάμεσα σε πρωτόκολλο εγγραφής-ενημέρωσης και πρωτόκολλο εγγραφής-ακύρωσης δεν είναι απλή. Τα πρωτόκολλα εγγραφής-ενημέρωσης απαιτούν την εκπομπή της νέας τιμής του δεδομένου σε κάθε τροποποίηση. Οι κρυφές μνήμες που ενημερώνονται έχουν έτσι ανά πάσα στιγμή, την πιο πρόσφατη τιμή του. Οι αντίστοιχοι επεξεργαστές μπορούν να λάβουν το δεδομένο χωρίς τη μεσολάβηση του διαύλου, κατευθείαν από την κρυφή μνήμη τους. Στα πρωτόκολλα εγγραφής-ακύρωσης, όμως, κάθε επεξεργαστής που έχει ακυρώσει το ιδιωτικό του αντίγραφο, θα προκαλέσει αστοχία όταν θελήσει να το προσπελάσει και η κρυφή μνήμη του θα καταφύγει στο δίαυλο για να παραλάβει από αλλού την πιο πρόσφατη τιμή του δεδομένου, αυξάνοντας την κίνηση και προκαλώντας καθυστερήσεις.

Στα πρωτόκολλα εγγραφής-ακύρωσης από την άλλη, η βασική προϋπόθεση για τροποποίηση ενός δεδομένου είναι η κατοχή της *αποκλειστικότητας* (exclusive read/write). Η ακύρωση ενός δεδομένου στις υπόλοιπες κρυφές μνήμες συμβαίνει τη στιγμή που κάποια κρυφή ζητήσει την αποκλειστικότητα του δεδομένου. Η αποκλειστικότητα παραμένει στην κρυφή μνήμη που τη ζήτησε, μέχρι κάποιος άλλος επεξεργαστής να ζητήσει το δεδομένο. Όσο διατηρεί την αποκλειστικότητα, ο επεξεργαστής μπορεί να τροποποιεί ανενόχλητα το δεδομένο πολλές φορές στην κρυφή μνήμη του, χωρίς να δημιουργεί επιπλέον κίνηση στον δίαυλο. Αντίθετα, τα πρωτόκολλα εγγραφής-ενημέρωσης απαιτούν τη χρήση του πολύτιμου διαύλου σε *κάθε* εγγραφή, ακόμα και αν οι εγγραφές γίνονται συνεχώς από τον ίδιο επεξεργαστή.

Καμία από τις δύο πολιτικές δεν μπορεί να χαρακτηριστεί ως η καλύτερη σε όλες τις περιπτώσεις. Τα χαρακτηριστικά των εφαρμογών που θα εκτελεστούν παίζουν μεγάλο ρόλο στη σχετική απόδοση των δύο μεθόδων. Πάντως, έχει γίνει αποδεκτό ότι τα πρωτόκολλα



Σχήμα 2.13 *Διάγραμμα καταστάσεων για το Παράδειγμα 2.2.*

εγγραφής-ενημέρωσης προκαλούν, σε γενικές γραμμές, μεγαλύτερη κίνηση στο δίαυλο. Αυτός είναι και ο λόγος που τα περισσότερα πολυεπεξεργαστικά συστήματα κοινόχρηστης μνήμης χρησιμοποιούν πρωτόκολλα εγγραφής-ακύρωσης.

Με δύο απλά πρωτόκολλα εγγραφής-ακύρωσης θα ασχοληθούμε και εμείς στα επόμενα παραδείγματα. Θα ξεκινήσουμε τη γνωριμία μας με ένα απλό πρωτόκολλο για κρυφές μνήμες που χρησιμοποιούν διεγγραφή. Η χρήση διεγγραφής οδηγεί μεν σε απλούστερες υλοποιήσεις, είναι όμως ιδιαίτερα απαιτητική στη χρήση του διαύλου, όπως γνωρίζουμε. Έτσι, στο δεύτερο παράδειγμα θα δούμε κάποιο πρωτόκολλο που υποθέτει τη χρήση υστεροεγγραφής.

Παράδειγμα 2.2

» *Πρωτόκολλο εγγραφής-ακύρωσης για κρυφές μνήμες με διεγγραφή*

Τα πρωτόκολλα συνοχής συνήθως περιγράφονται ως ένα διάγραμμα καταστάσεων. Το διάγραμμα αυτό αναφέρεται στις δυνατές καταστάσεις στις οποίες μπορεί να βρεθεί ένα συγκεκριμένο δεδομένο σε μία συγκεκριμένη κρυφή μνήμη. Η μετάβαση από κατάσταση σε κατάσταση γίνεται υπό την επήρεια γεγονότων, όπως π.χ. τροποποίηση του δεδομένου. Υποτίθεται ότι ακριβώς η ίδια μηχανή καταστάσεων είναι σε ενέργεια για κάθε δεδομένο, σε κάθε κρυφή μνήμη στο σύστημα.

Μία κρυφή μνήμη δέχεται δύο ειδών ερεθίσματα τα οποία είναι ικανά να αλλάξουν την κατάσταση ενός δεδομένου. Το πρώτο είδος είναι οι αιτήσεις που προέρχονται από τον επεξεργαστή, οι οποίες μπορεί να είναι για ανάγνωση (R) ή εγγραφή (W) του δεδομένου. Το δεύτερο είδος είναι οι αιτήσεις που παρατηρεί η κρυφή μνήμη στο δίαυλο και οι οποίες προέρχονται από άλλες κρυφές μνήμες. Προκειμένου να γίνεται εύκολα ο διαχωρισμός, στα διαγράμματα που θα δούμε, οι αιτήσεις που παρατηρούνται στο δίαυλο σημειώνονται σε γκρι φόντο. Γκρι χρώμα χρησιμοποιείται, επίσης, και για τις μεταβάσεις καταστάσεων που οφείλονται σε τέτοιες αιτήσεις

(δείτε και το Σχ. 2.13). Δείτε, επίσης, το Πρόβλημα 2.4 για ένα τρίτο είδος ερεθίσματος, προερχόμενο από την ίδια τη κρυφή μνήμη.

Στο Σχ. 2.13 φαίνεται ένα απλό πρωτόκολλο εγγραφής-ακύρωσης για κρυφή μνήμη που χρησιμοποιεί διεγγραφή. Το δεδομένο μπορεί να βρίσκεται σε δύο δυνατές καταστάσεις: έγκυρο (valid) και άκυρο (invalid). Το δεδομένο είναι άκυρο εφόσον κάποιος άλλος επεξεργαστής το έχει τροποποιήσει, χωρίς να έχουμε λάβει την νέα του τιμή, ενώ είναι έγκυρο σε κάθε άλλη περίπτωση. Όσο το δεδομένο είναι έγκυρο, ο επεξεργαστής μπορεί να το διαβάζει (R) ή να το τροποποιεί (W) ανενόχλητα. Στην περίπτωση που το τροποποιήσει, θα πρέπει να ενημερώσει κατευθείαν την κύρια μνήμη για τη νέα τιμή του δεδομένου (διεγγραφή), προκαλώντας, έτσι, έναν κύκλο εγγραφής στο δίαυλο (BusWrite). Στο σχήμα, οι ενέργειες που προκαλεί κάποια αίτηση σημειώνονται με πλάγια γράμματα. Σημειώστε ότι αυτός ο κύκλος εγγραφής είναι που αναγκάζει κάθε άλλη κρυφή μνήμη, που διαθέτει αντίγραφο του δεδομένου, να το ακυρώσει.

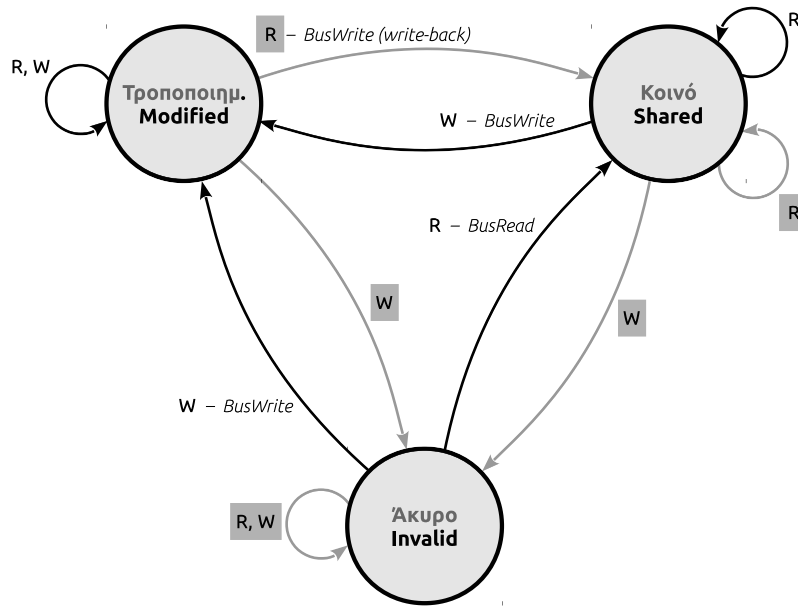
Το δεδομένο ακυρώνεται μόνο όταν υπάρξει στο δίαυλο αίτηση εγγραφής του (W σε γκρι φόντο). Αυτό σημαίνει ότι κάποια άλλη κρυφή μνήμη τροποποιεί το δεδομένο και επομένως, το αντίγραφο που διαθέτουμε δεν είναι, πλέον, έγκυρο. Αφού ακυρωθεί το δεδομένο, οποιαδήποτε προσπάθεια από τον επεξεργαστή να το διαβάσει ή να το γράφει, οδηγεί σε αντίστοιχη αίτηση προς το δίαυλο για την παραλαβή ή αποστολή της νέας του τιμής. Ταυτόχρονα, θα υπάρξει και αλλαγή της κατάστασης σε έγκυρη.

Παράδειγμα 2.3

» Πρωτόκολλο MSI για κρυφές μνήμες με υστεροεγγραφή

Στο Σχ. 2.14 φαίνεται το διάγραμμα καταστάσεων ενός πρωτοκόλλου εγγραφής-ακύρωσης για κρυφές που χρησιμοποιούν υστεροεγγραφή. Πρόκειται για ένα αρκετά απλό πρωτόκολλο, στο οποίο ένα δεδομένο μπορεί να βρεθεί σε τρεις καταστάσεις: *τροποποιημένο* (**m**odified), *κοινό* (**s**hared) και *άκυρο* (**i**nvalid). Γι' αυτό και το συναντά κανείς (σε διάφορες παραλλαγές) και ως πρωτόκολλο MSI.

Ένα δεδομένο είναι στην κοινή κατάσταση, όταν σε όλες τις κρυφές μνήμες που διαθέτουν αντίγραφά του, έχει την ίδια τιμή. Δηλαδή, υπάρχουν πολλά αλλά πανομοιότυπα αντίγραφα του δεδομένου. Αντίθετα, η τροποποιημένη κατάσταση δηλώνει ότι μόνο η συγκεκριμένη κρυφή μνήμη διαθέτει το πιο πρόσφατο αντίγραφο (αποκλειστικότητα)· όλες οι άλλες το έχουν ακυρώσει, ενώ και η κύρια μνήμη δεν γνωρίζει τη νέα τιμή. Σε κάθε άλλη περίπτωση, για τη δεδομένη κρυφή μνήμη, το δεδομένο



Σχήμα 2.14 *Διάγραμμα καταστάσεων για το Παράδειγμα 2.3.*

θεωρείται άκυρο.

Ξεκινώντας από ένα άκυρο αντίγραφο, όταν ο επεξεργαστής ζητήσει να το διαβάσει, θα προκληθεί κύκλος ανάγνωσης στο δίαυλο, προκειμένου να ληφθεί το σωστό δεδομένο. Ταυτόχρονα, το δεδομένο γίνεται πλέον κοινό. Προσέξτε ότι στο σημείο αυτό υπάρχουν διάφορες επιλογές υλοποίησης. Για παράδειγμα, την πιο πρόσφατη τιμή του δεδομένου μπορεί να την εναποθέσει στο δίαυλο η κρυφή μνήμη που την έχει ή μπορεί να την δώσει η κύρια μνήμη, εφόσον, όμως, έχει ενημερωθεί (έχει προηγηθεί, δηλαδή, κάποια φάση υστεροεγγραφής).

Το δεδομένο παραμένει στην κοινή κατάσταση και δίνεται αμέσως στον επεξεργαστή όταν αυτός το ζητήσει για ανάγνωση, χωρίς αίτηση στο δίαυλο. Επίσης, οποιοσδήποτε άλλος επεξεργαστής και να το διαβάσει, η κατάσταση παραμένει κοινή. Αλλαγή κατάστασης θα γίνει μόνο όταν (α) ο επεξεργαστής ζητήσει εγγραφή—οπότε οδηγούμαστε στην τροποποιημένη κατάσταση—ή (β) κάποιος άλλος επεξεργαστής ζητήσει εγγραφή—οπότε ακυρώνεται το αντίγραφο που διαθέτουμε.

Κατά τη μεταβίβαση από την κοινή κατάσταση στην τροποποιημένη (μετά από αίτηση εγγραφής του επεξεργαστή), είναι απαραίτητο να προκληθεί κύκλος εγγραφής στο δίαυλο BusWrite), προκειμένου να ενημερωθούν όλες οι υπόλοιπες κρυφές μνήμες και να ακυρώσουν το αντίγραφό τους. Όμως, αφού το δεδομένο εισέλθει στην τροποποιημένη κατάσταση, ο επεξεργαστής αποκτά την αποκλειστικότητα σε αυτό και μπορεί να το διαβάζει και να το τροποποιεί χωρίς να δημιουργεί την παραμικρή κίνηση στο δίαυλο.

Από την τροποποιημένη κατάσταση εξέρχεται το δεδομένο μόνο μετά από αίτηση στο δίαυλο που έγινε από άλλον επεξεργαστή: είτε αίτηση εγγραφής (οπότε το δεδομένο μας είναι, πλέον, άκυρο) είτε αίτηση ανάγνωσης (οπότε το δεδομένο θα γίνει, αναγκαστικά, κοινό). Στη δεύτερη περίπτωση, ο επεξεργαστής θα πρέπει (ανάλογα, πάντα, με τις λεπτομέρειες της υλοποίησης) να κάνει υστεροεγγραφή στην κύρια μνήμη, ώστε αυτή να λάβει την πιο πρόσφατη τιμή του δεδομένου. Κατόπιν θα την λάβει, με τη σειρά του, ο άλλος επεξεργαστής που έκανε την αίτηση ανάγνωσης.

Δεν έχουμε σκοπό να εμβαθύνουμε περισσότερο στα πρωτόκολλα παρακολούθησης. Θα πρέπει, πάντως, να έχει γίνει φανερό ότι γίνονται γρήγορα ιδιαίτερα πολύπλοκα, όταν προσπαθήσει κανείς να κάνει βελτιώσεις προκειμένου να ελαχιστοποιείται η χρήση του διαύλου. Στην πράξη, πολλές μηχανές (για παράδειγμα επεξεργαστές της Intel) χρησιμοποιούν παραλλαγές ενός πρωτοκόλλου εγγραφής-ακύρωσης γνωστού ως MESI ή *πρωτόκολλο Illinois*, το οποίο έχει μία παραπάνω κατάσταση από το πρωτόκολλο MSI στο Παράδειγμα 2.3. Η κατάσταση αυτή («αποκλειστική»—**e**xclusive) δηλώνει ότι η κρυφή μνήμη έχει το μοναδικό αντίγραφο σε όλο το σύστημα (όπως στην τροποποιημένη κατάσταση), το οποίο, όμως, δεν έχει τροποποιηθεί. Από την αποκλειστική κατάσταση υπάρχει μετάβαση προς την τροποποιημένη, αν ο επεξεργαστής τροποποιήσει το δεδομένο *χωρίς όμως καμία ειδοποίηση/κίνηση στον δίαυλο*, ενώ υπάρχει μετάβαση προς την κοινή κατάσταση, αν κάποιος άλλος επεξεργαστής διαβάσει το δεδομένο. Δείτε επίσης το Πρόβλημα 2.5.

Στους επεξεργαστές UltraSparc, η εταιρεία Sun χρησιμοποίησε ένα πρωτόκολλο με μία ακόμα κατάσταση (O), το οποίο ονομάζει MOESI. Το ίδιο πρωτόκολλο χρησιμοποιεί και η εταιρεία AMD στους επεξεργαστές Athlon και Opteron. Στην κατάσταση O γίνεται μετάβαση από την τροποποιημένη κατάσταση, όταν μία άλλη κρυφή μνήμη ζητήσει να διαβάσει το δεδομένο αλλά η κύρια μνήμη δεν έχει ενημερωθεί για τη νέα τιμή του. Η μετάβαση από την τροποποιημένη στην κοινή κατάσταση που έχουμε στο Σχ. 2.14, προϋποθέτει ότι η κύρια μνήμη έχει ενημερωθεί (μέσω υστεροεγγραφής). Πρόκειται, επομένως, για μία κατάσταση παρόμοια με την κοινή, μόνο που η κύρια μνήμη δεν γνωρίζει τη σωστή τιμή (κοινή-τροποποιημένη—shared-m**o**dified). Ουσιαστικά, ο επεξεργαστής που είχε τροποποιήσει το δεδομένο, μεταπίπτει στην κατάσταση O και είναι πλέον υπεύθυνος να παρέχει τη σωστή τιμή του δεδομένου προς όποια άλλη κρυφή μνήμη την θελήσει. Με αυτόν τον τρόπο, απαλλάσσεται εντελώς η μνήμη από την υποχρέωση να διατηρείται ενήμερη, παρακολουθώντας συνεχώς τον δίαυλο για τυχόν εγγραφές.

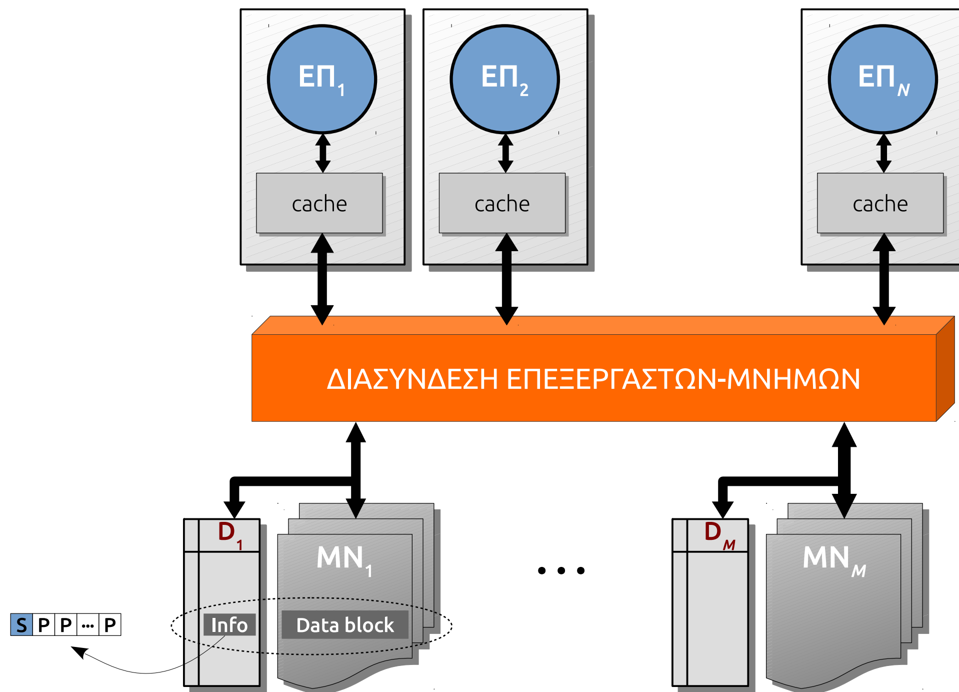
2.6 Πρωτόκολλα καταλόγων

Τα πρωτόκολλα παρακολούθησης προϋποθέτουν την ύπαρξη ενός κοινού μέσου όπου εμφανίζονται όλες οι αιτήσεις από / προς τη μνήμη, και από όπου κάθε κρυφή μνήμη μπορεί να παρακολουθεί την κίνηση των δεδομένων που την ενδιαφέρουν. Κάτι τέτοιο, όμως, δεν είναι δυνατόν να συμβεί όταν το δίκτυο δεν είναι δίαυλος. Στην περίπτωση αυτή, μία κύρια μνήμη συνδέεται μέσω διαφορετικής διαδρομής με κάθε επεξεργαστή και επομένως, μπορεί να συνεννοηθεί μόνο με έναν από αυτούς τη φορά. Εάν απαιτείται να ενημερώσει όλους τους επεξεργαστές για κάτι, είναι πολύ πιθανό να απαιτηθούν πολύ κύκλοι ιδιωτικών επικοινωνιών.

Ως παράδειγμα, φανταστείτε ότι διαθέτουμε ένα δίκτυο Δέλτα, και ότι η συνοχή των κρυφών μνημών εξασφαλίζεται κάνοντας χρήση του πρωτοκόλλου στο Παράδειγμα 2.3, χωρίς καμία τροποποίηση. Εάν ένα δεδομένο σε μία κρυφή μνήμη X μεταβεί από την κοινή στην τροποποιημένη κατάσταση, θα πρέπει όλες οι υπόλοιπες κρυφές μνήμες να ενημερωθούν, ώστε να ακυρώσουν το αντίγραφό τους. Από τη στιγμή που δεν υπάρχει πληροφορία για το ποιες κρυφές μνήμες έχουν αντίγραφο του δεδομένου, η συνοχή μπορεί να εξασφαλιστεί μόνο ως εξής: η κρυφή μνήμη X επικοινωνεί μέσω του δικτύου Δέλτα με την κύρια μνήμη, όπου φυλάσσεται το δεδομένο. Αυτή με τη σειρά της επικοινωνεί με όλες τις υπόλοιπες κρυφές μνήμες, μία προς μία, στέλνοντάς τους το σήμα ακύρωσης. Μπορεί εύκολα να καταλάβει κανείς πόσο χρονοβόρα είναι αυτή η διαδικασία. Και όχι μόνο αυτό! Όσο αυξάνει ο αριθμός των επεξεργαστών, τόσο περισσότερες επικοινωνίες θα χρειάζονται. Ας το σκεφτούμε αυτό λίγο. Ο λόγος που καταφύγαμε σε άλλα δίκτυα διασύνδεσης είναι ότι ο δίαυλος δεν κλιμακώνει σωστά τις επιδόσεις του όταν αυξάνει ο αριθμός των επεξεργαστών. Όμως, βλέπουμε ότι κινδυνεύει να συμβεί το ίδιο και στα υπόλοιπα δίκτυα, αν το πρωτόκολλο συνοχής δεν είναι σωστά προσαρμοσμένο.

Το κλειδί για ένα επιτυχημένο πρωτόκολλο συνοχής είναι να περιορίσει στο ελάχιστο τις επικοινωνίες που απαιτούνται. Αυτό μπορεί να γίνει μόνο αν οι επικοινωνίες περιορίζονται μεταξύ των ενδιαφερομένων επεξεργαστών / μνημών, αποφεύγοντας την εμπλοκή όλων των υπολοίπων. Για να γίνει κάτι τέτοιο, θα πρέπει με κάποιο τρόπο να υπάρχει η πληροφορία για το ποιος επεξεργαστής έχει στην κρυφή μνήμη του ποιο δεδομένο. Χρειάζονται, επομένως, *κατάλογοι* (directories)που διατηρούν αυτές τις πληροφορίες. Τα πρωτόκολλα που τους χρησιμοποιούν λέγονται *πρωτόκολλα καταλόγων* (directory-based protocols).

Μία απλή ιδέα θα ήταν να αποθηκεύονται όλες αυτές οι πληροφορίες σε μία ειδική μνήμη και να την συμβουλεύονται όλοι, όταν χρειάζεται. Αυτή είναι η μέθοδος *κεντρικού καταλόγου* (central directory), η οποία, όμως, δεν ενδείκνυται, αφού είναι φανερό ότι η συγκεκριμένη μνήμη πρώτον, θα δέχεται καταιγισμό αιτήσεων και δεύτερον, θα αποτελεί κρίσιμο σημείο του συστήματος. Οι μόνες βιώσιμες στρατηγικές είναι αυτές με *κατανεμημένους* καταλόγους (distributed directories), όπου η κάθε μνήμη έχει δικό της κατάλογο. Εκεί φυλάσσεται, για κάθε δεδομένο, μία λίστα με τους επεξεργαστές που διαθέτουν αν-



Σχήμα 2.15 *Οργάνωση με καταλόγους.*

τίγραφο στην κρυφή μνήμη τους. Στο Σχ. 2.15 φαίνεται η φυσική οργάνωση για μία τέτοια στρατηγική.

Κάθε μνήμη MN$_i$ διαθέτει δικό της κατάλογο D_i, ο οποίος είναι ουσιαστικά μία εξειδικευμένη μνήμη που φυλάει συγκεκριμένες πληροφορίες. Οι πληροφορίες αυτές περιλαμβάνουν κάποια πεδία *παρουσίας* P (presence info), καθένα από τα οποία αναφέρει έναν επεξεργαστή ο οποίος διαθέτει αντίγραφο από το δεδομένο. Προσέξτε ότι, επειδή οι κρυφές μνήμες επικοινωνούν με την κύρια μνήμη σε μεγέθη γραμμών (cache lines), όταν λέμε «δεδομένο», εδώ, εννοούμε ένα ολόκληρο μπλοκ μνήμης ή ισοδύναμα μία ολόκληρη γραμμή κρυφής μνήμης. Εκτός από τα πεδία παρουσίας, υπάρχει ένα ακόμα πεδίο (S στο Σχ. 2.15), συνήθως ενός μόνο bit, το οποίο υποδηλώνει την κατάσταση του δεδομένου. Συγκεκριμένα, μπορεί να είναι *καθαρό* (clean, C) που σημαίνει ότι κανένας επεξεργαστής δεν έχει τροποποιήσει το αντίγραφό του ή *ακάθαρτο* (dirty, D), σε αντίθετη περίπτωση. Όταν το δεδομένο είναι ακάθαρτο, το μόνο πεδίο παρουσίας που θα υπάρχει είναι αυτό που αναφέρει την κρυφή μνήμη με το τροποποιημένο αντίγραφο. Όλες οι υπόλοιπες θα πρέπει να έχουν ακυρώσει το αντίγραφό τους.

Παράδειγμα 2.4

» *Πρωτόκολλο με πλήρεις καταλόγους—full-map directories*

Στο παράδειγμα αυτό, θα δούμε πώς λειτουργεί ένα απλό πρωτόκολλο καταλόγων. Αν υπάρχουν N επεξεργαστές στο σύστημα, τότε οι καταχωρήσεις στον κατάλογο

θα αποτελούνται από N πεδία παρουσίας του 1 bit. Το i-οστό bit θα έχει την τιμή 1, αν ο επεξεργαστής i διαθέτει αντίγραφο του δεδομένου στην κρυφή μνήμη του. Στην πρώτη σκηνή, στο Σχ. 2.16, οι επεξεργαστές 1 και 2 ζητούν το δεδομένο προς ανάγνωση. Στη δεύτερη σκηνή, μπορεί να δει κανείς πως τα αντίστοιχα δύο bits στον κατάλογο έχουν πάρει την τιμή 1.

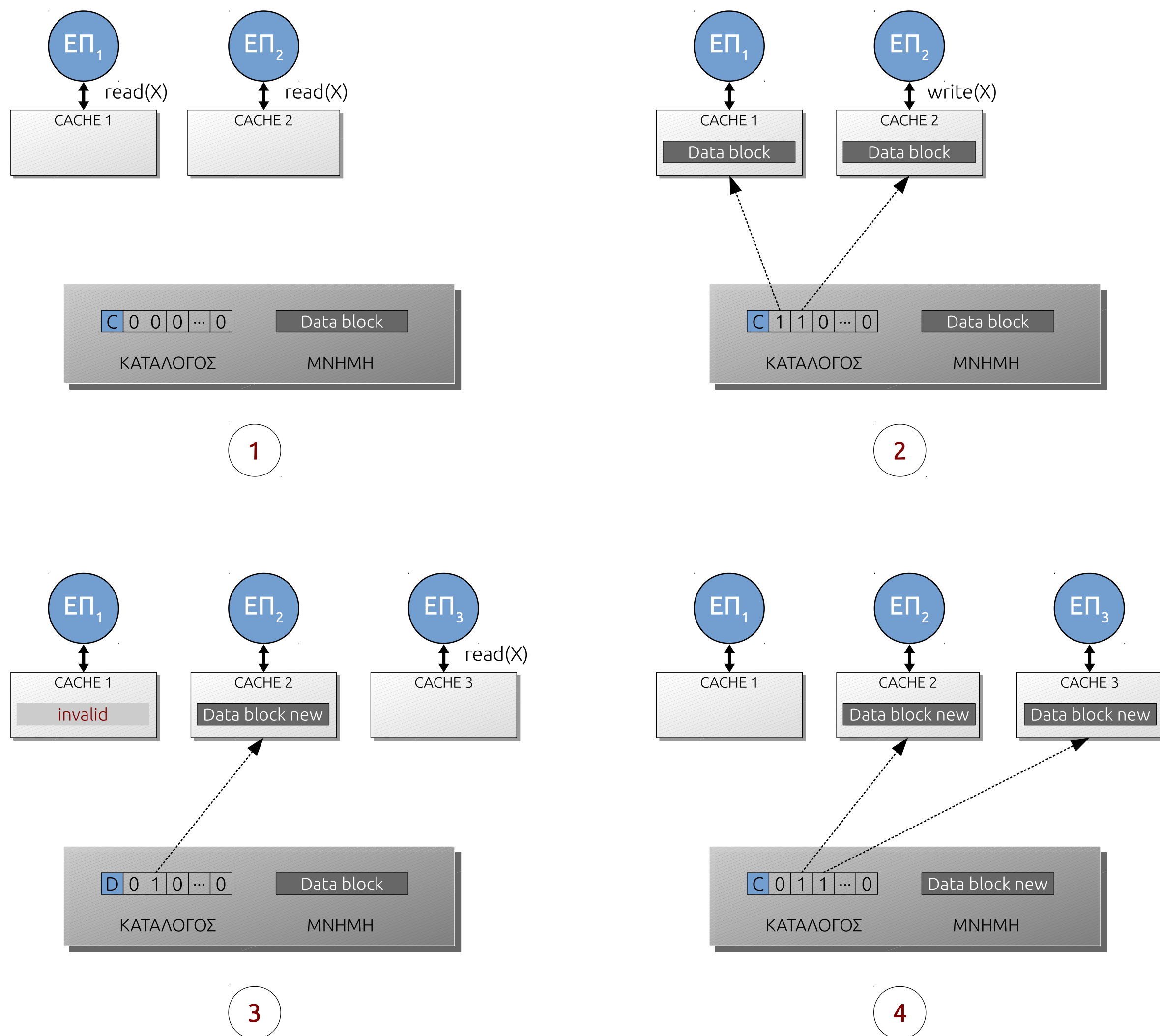
Στη δεύτερη σκηνή, ο δεύτερος επεξεργαστής ζητά να τροποποιήσει το δεδομένο. Αυτό που θα γίνει και το οποίο δεν φαίνεται στο Σχ. 2.16 είναι το εξής: πριν την τροποποίηση, υπάρχει επικοινωνία με τη μνήμη από την οποία ήρθε το δεδομένο. Η μνήμη αυτή, κοιτώντας τον κατάλογο βρίσκει τους υπόλοιπους επεξεργαστές που έχουν αντίγραφο (ο επεξεργαστής 1 στο παράδειγμά μας) και τους στέλνει μήνυμα ακύρωσης του δεδομένου τους. Αυτοί, με τη σειρά τους, μόλις το ακυρώσουν ενημερώνουν τη μνήμη για την ολοκλήρωση της ακύρωσης. Αφού η μνήμη λάβει μηνύματα ολοκλήρωσης από όλους τους εμπλεκόμενους επεξεργαστές, τότε δίνει την άδεια στον επεξεργαστή 2 να τροποποιήσει το δεδομένο στην κρυφή μνήμη του.

Το αποτέλεσμα φαίνεται στην τρίτη σκηνή, όπου η κρυφή μνήμη 1 έχει ακυρώσει το δεδομένο της και ο κατάλογος έχει δηλώσει το δεδομένο ως ακάθαρτο (D), με μοναδικό κάτοχο του σωστού δεδομένου την κρυφή μνήμη 2.

Αν, τώρα, ένας άλλος επεξεργαστής (π.χ. ο 3) ζητήσει από τη μνήμη να διαβάσει το δεδομένο, η μνήμη ελέγχοντας τον κατάλογο θα διαπιστώσει ότι δεν διαθέτει τη σωστή τιμή. Από τον κατάλογο θα δει ότι η κρυφή μνήμη 2 είναι αυτή που έχει το σωστό δεδομένο. Θα το ζητήσει, επομένως, από αυτήν (υστεροεγγραφή) και τέλος, θα το δώσει στον επεξεργαστή 3 που το ζήτησε. Η τελική κατάσταση απεικονίζεται στη σκηνή 4.

Το βασικό μειονέκτημα που έχει το πρωτόκολλο του προηγούμενου παραδείγματος είναι οι απαιτήσεις του σε χώρο αποθήκευσης των καταλόγων, κάτι το οποίο μπορεί να δει κανείς έμπρακτα αν προσπαθήσει να λύσει το Πρόβλημα 2.6. Η κατάσταση, μάλιστα, χειροτερεύει με την αύξηση των επεξεργαστών.

Προκειμένου να βελτιωθεί η κατάσταση, μία λύση είναι να αυξηθεί το μέγεθος του μπλοκ μνήμης και αντίστοιχα της γραμμής της κρυφής μνήμης. Αυτό, σίγουρα, επιδρά θετικά ως προς τον χώρο αποθήκευσης, όμως επιβαρύνει τις επικοινωνίες, αφού θα πρέπει να μετακινούνται περισσότερα δεδομένα κάθε φορά μεταξύ κύριας και κρυφής μνήμης. Μία δεύτερη λύση είναι να μειωθεί το μέγεθος των καταχωρήσεων στους καταλόγους. Συγκεκριμένα, βάζουμε τον περιορισμό ότι το πολύ K επεξεργαστές τη φορά μπορεί να έχουν έγκυρο αντίγραφο του κάθε δεδομένου. Έτσι, η κάθε καταχώρηση δεν χρειάζεται να έχει πεδία παρουσίας για N επεξεργαστές, παρά μόνο για $K < N$ από αυτούς. Τέτοια πρωτόκολλα ονομάζονται *περιορισμένων καταλόγων* (limited directories). Το επόμενο



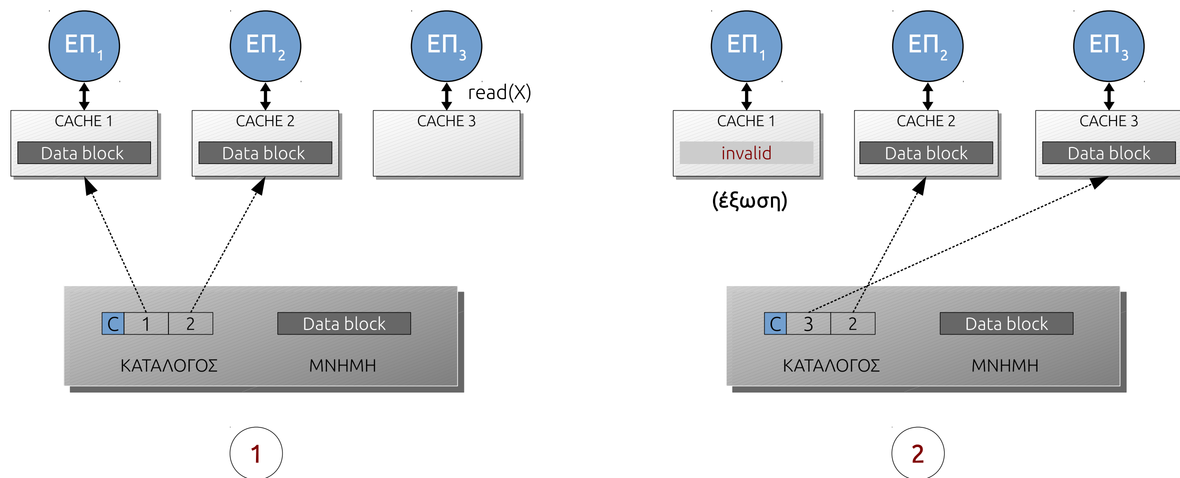
Σχήμα 2.16 *Πρωτόκολλο με πλήρεις καταλόγους.*

παράδειγμα περιγράφει τη λειτουργία τους.

Παράδειγμα 2.5

» *Πρωτόκολλο με περιορισμένους καταλόγους—limited directories*

Αν οι καταχωρήσεις στον κατάλογο διαθέτουν $K < N$, το πολύ, πεδία παρουσίας, αυτά δεν μπορεί πλέον να έχουν μέγεθος 1 bit. Κάθε πεδίο θα πρέπει να περιέχει τον αριθμό του επεξεργαστή, που διαθέτει αντίγραφο του δεδομένου. Έτσι, αν οι επεξεργαστές είναι αριθμημένοι από 0 έως $N-1$, θα απαιτηθούν $\log_2 N$ bits σε κάθε πεδίο P του Σχ. 2.15. Σε κάθε ένα από τα K πεδία παρουσίας, θα καταχωρείται ο αριθμός του επεξεργαστή που διαθέτει το δεδομένο. Το bit κατάστασης, καθώς και όλη η λειτουργία του καταλόγου θα είναι ίδια με το Παράδειγμα 2.4, όπως φαίνεται και στην πρώτη σκηνή του Σχ. 2.17, όπου υποθέτουμε ότι $K = 2$.



Σχήμα 2.17 *Πρωτόκολλο με περιορισμένους καταλόγους.*

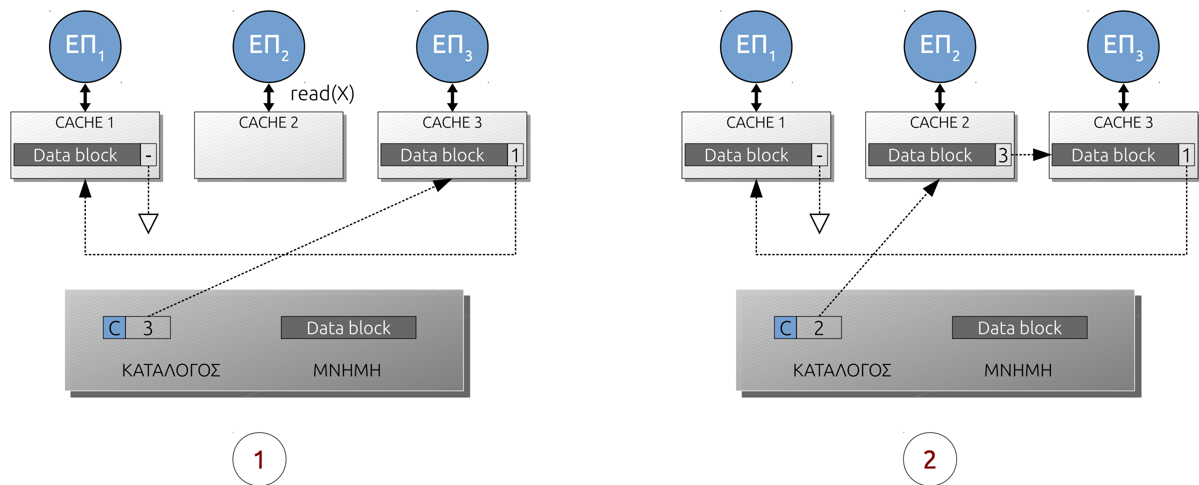
Το μόνο πρόβλημα που παρουσιάζεται εδώ, είναι τι γίνεται στην περίπτωση που παραπάνω από K επεξεργαστές ζητούν το δεδομένο. Στο Σχ. 2.17, εμφανίζεται μία αίτηση ανάγνωσης του δεδομένου από τον επεξεργαστή 3, τη στιγμή που αντίγραφα έχουν δοθεί στις κρυφές μνήμες 1 και 2. Αυτό που θα συμβεί είναι ανάλογο με την αντικατάσταση ενός δεδομένου σε μία κρυφή μνήμη. Συγκεκριμένα, θα επιλεχθεί ένας από τους επεξεργαστές που διαθέτουν το αντίγραφο και θα του σταλεί μήνυμα *ακύρωσης* του δεδομένου του. Έτσι, θα δημιουργηθεί μία κενή θέση στον κατάλογο, όπου μπορεί να καταχωρηθεί ο επεξεργαστής 3 μόλις του σταλεί το δεδομένο (δεύτερη σκηνή στο Σχ. 2.17). Η ενέργεια αυτή είναι γνωστή ως *έξωση* (eviction).

Στα πρωτόκολλα με περιορισμένους καταλόγους θα πρέπει να επιλεγεί με προσοχή η τιμή του K. Από τη μία, το K δεν πρέπει να είναι πολύ μεγάλο, ώστε οι κατάλογοι να μην πιάνουν πολύ χώρο. Από την άλλη, το K δεν θα πρέπει να είναι πολύ μικρό, ώστε να μη συμβαίνει συχνά το ανεπιθύμητο φαινόμενο της έξωσης. Στην πράξη έχει βρεθεί ότι το 5 ή το 6 είναι μία πολύ καλή τιμή για το K.

Μία άλλη στρατηγική για τον περιορισμό του μεγέθους των καταλόγων είναι τα λεγόμενα *αλυσιδωτά πρωτόκολλα* (chained directories). Η βασική ιδέα είναι να μην γνωρίζει ο κατάλογος όλους τους επεξεργαστές που διαθέτουν αντίγραφο του δεδομένου, παρά μόνο έναν από αυτούς. Στην κρυφή μνήμη του επεξεργαστή αυτού θα υπάρχει πληροφορία για το ποιος είναι ο επόμενος επεξεργαστής που διαθέτει αντίγραφο κ.ο.κ. Η πληροφορία, δηλαδή, θα είναι διαμοιρασμένη στις κρυφές μνήμες, οι οποίες θα σχηματίζουν έτσι, για κάθε δεδομένο, μία διαφορετική λίστα μεταξύ τους.

Παράδειγμα 2.6

» *Πρωτόκολλο με απλά συνδεδεμένους αλυσιδωτούς καταλόγους—singly-linked chained directories*



Σχήμα 2.18 *Πρωτόκολλο με αλυσιδωτούς καταλόγους.*

Μία απλή υλοποίηση αλυσιδωτού πρωτοκόλλου φαίνεται στο Σχ. 2.18. Ο κατάλογος στην κύρια μνήμη περιέχει τον αριθμό (3) ενός μόνο από τους επεξεργαστές που διαθέτουν το δεδομένο. Στην κρυφή μνήμη του επεξεργαστή αυτού, μαζί με το δεδομένο, φυλάσσεται πληροφορία που δείχνει έναν ακόμα επεξεργαστή που διαθέτει αντίγραφο (1). Επειδή δεν υπάρχει άλλος επεξεργαστής με αντίγραφο του δεδομένου, η κρυφή μνήμη του επεξεργαστή 1 δεν δείχνει πουθενά. Σχηματίζεται έτσι, μία συνδεδεμένη λίστα μεταξύ των κρυφών μνημών που περιέχουν το δεδομένο. Ο κατάλογος στην κύρια μνήμη δείχνει απλά την αρχή της λίστας.

Στην πρώτη σκηνή, στο Σχ. 2.18, εμφανίζεται μία αίτηση για ανάγνωση του δεδομένου από τον επεξεργαστή 2. Η κύρια μνήμη στέλνει το δεδομένο στον επεξεργαστή 2, αλλά μαζί με το δεδομένο στέλνει και τον αριθμό του επεξεργαστή στην αρχή της λίστας. Έτσι, η κρυφή μνήμη 2 γίνεται η κορυφή της λίστας, δείχνοντας προς τον επεξεργαστή 3.

Τα πρωτόκολλα με αλυσιδωτούς καταλόγους είναι θεωρητικά βέλτιστα από άποψη χώρου, αφού η λίστα κρατά τόσες πληροφορίες όσα ακριβώς είναι και τα αντίγραφα του δεδομένου. Όμως, γίνονται αρκετά πολύπλοκα όταν συμβεί, π.χ. ακύρωση του δεδομένου ή αντικατάστασή του σε μία από τις κρυφές μνήμες. Ας υποθέσουμε για παράδειγμα, ότι στο Σχ. 2.18, σκηνή 2, η κρυφή μνήμη 3 κάνει αντικατάσταση του δεδομένου. Πώς βγάζει τον εαυτό της εκτός λίστας; Μία λύση είναι να ενημερώσει την κύρια μνήμη, η οποία θα πρέπει να στείλει στη λίστα ειδικό μήνυμα με δείκτη στον επεξεργαστή 1, ώστε η κρυφή μνήμη 2 (που προηγείται της 3 στη λίστα) να αλλάξει τον δείκτη της. Μία άλλη λύση είναι να μην εμπλακεί η κύρια μνήμη και να σταλεί στο τμήμα της αλυσίδας από την κρυφή μνήμη 3 και μετά, ένα μήνυμα ακύρωσης του δεδομένου. Όσο και αν φαίνεται παράξενο, η

δεύτερη λύση είναι πολλές φορές καλύτερη από την πρώτη.

Το πρόβλημα μπορεί να λυθεί ευκολότερα αν αντί για απλά συνδεδεμένη χρησιμοποιούσαμε *διπλά συνδεδεμένη* λίστα. Σε κάθε κρυφή μνήμη, μαζί με το δεδομένο και τον δείκτη προς την επόμενη κρυφή μνήμη θα υπήρχε και ένας επιπλέον δείκτης προς την προηγούμενη κρυφή μνήμη. Δεν είναι δύσκολο να φανταστεί κανείς πόσο πολύπλοκη γίνεται, έτσι, η όλη λειτουργία της συνοχής. Ένα τέτοιο πρωτόκολλο είναι το SCI (Scalable Coherency Interface) της IEEE, το οποίο το έχουν χρησιμοποιήσει εμπορικοί πολυεπεξεργαστές (όπως οι Sequent Numa-Q και Convex Exemplar). Προκειμένου να ξεπεραστούν οι ιδιαίτερα πολύπλοκες λεπτομέρειές του, εκτός από λεπτομερή περιγραφή, περιλαμβάνει ακόμα και κώδικα σε γλώσσα C που υλοποιεί τη λειτουργία του!

2.7 Συνέπεια μνήμης

Έχοντας μελετήσει τα περί συνοχής της κρυφής μνήμης, σιγουρευτήκαμε ότι ο νέος παράλληλος υπολογιστής που αγοράσαμε πράγματι διαθέτει συνοχή. Δοκιμάσαμε λοιπόν να γράψουμε ένα απλό παράλληλο πρόγραμμα, όπως φαίνεται στο Σχ. 2.19, με δύο διεργασίες και δύο κοινόχρηστες μεταβλητές. Η διεργασία Δ1 εκτελείται σε έναν επεξεργαστή και θέτει την τιμή των δύο κοινών μεταβλητών, ενώ η Δ2 εκτελείται στον άλλο επεξεργαστή και απλά τυπώνει τις τιμές των μεταβλητών.

Αρχικά είχαμε `A = B = 0`

Δ1	**Δ2**
`...`	`...`
`A = 1;`	`printf("%d", B);`
`B = 1;`	`printf("%d", A);`

Σχήμα 2.19 *Δύο απλές διεργασίες.*

Εκτελούμε το πρόγραμμα και στην οθόνη μας τυπώνεται `10`! Κανονικά, αφού το `B` τυπώθηκε ως 1 από τη Δ2 και αφού το `B = 1` έγινε μετά το `A = 1` στη Δ1, είναι μάλλον προφανές ότι η Δ2 έπρεπε να τυπώσει ότι `A = 1` και να πάρουμε `11` ως εκτύπωση. Μήπως πρέπει να πάμε τον υπολογιστή για επισκευή;

Γενικά, θα μπορούσαμε να δεχτούμε εκτυπώσεις όπως:

`00` (αν η Δ2 τέλειωσε πριν αρχίσει καν η Δ1),
`01` (αν η Δ2 διάβασε το `B` πριν το `B = 1` αλλά μετά το `A = 1`),
`11` (αν η Δ1 τέλειωσε πριν αρχίσει η Δ2),

οι οποίες είναι πιθανές λόγω της ασύγχρονης εκτέλεσης των δύο διεργασιών. Το `10` όμως είναι αδιανόητο.

Προσέξτε ότι οι καταχωρήσεις «γράφουν» (τροποποιούν) κάτι στις μεταβλητές, ενώ το `printf` απλά «διαβάζει» τη μεταβλητή και την τυπώνει. Δηλαδή, το παραπάνω είναι ισοδύναμο με το εξής:

Δ1	**Δ2**
`...`	`...`
`write(A);`	`read(B);`
`write(B);`	`read(A);`

και ανακαλύψαμε ότι στο μηχάνημα η CPU που εκτελεί τη Δ2 διαβάζει το `A` *μετά* την εγγραφή του `B` από τη Δ1 και παρ' όλα αυτά, διαβάζει την παλιά τιμή του `A` και όχι την πιο πρόσφατη!

Η συνοχή της κρυφής μνήμης δεν μας βοηθάει σε αυτή την περίπτωση. Πιο συγκεκριμένα, τα πρωτόκολλα συνοχής εγγυώνται ότι τις τροποποιήσεις των κοινών μεταβλητών θα τις δουν όλοι οι ενδιαφερόμενοι (κρυφές μνήμες, κύρια μνήμη), αλλά δεν προσδιορίζουν το *πότε* και εκεί ακριβώς είναι το κρίσιμο σημείο. Τη σχετική σειρά με την οποία εμφανίζονται οι εγγραφές στην κοινόχρηστη μνήμη και στις διάφορες κρυφές μνήμες, την καθορίζει η *συνέπεια της μνήμης* (memory consistency).

Η συνέπεια της μνήμης είναι ένα συμβόλαιο μεταξύ του υλικού της μνήμης και του λογισμικού, ότι, εφόσον το λογισμικό ακολουθεί ορισμένους κανόνες, τότε η μνήμη θα εγγυάται συγκεκριμένα αποτελέσματα. Με πιο απλά λόγια, η συνέπεια της μνήμης προσδιορίζει πώς και με ποια σειρά θα εμφανίζονται στους επεξεργαστές οι εγγραφές που γίνονται προς τη μνήμη.

Όταν λέμε ότι μία εγγραφή «εμφανίζεται» ή «φαίνεται» σε έναν επεξεργαστή τί εννοούμε; Όταν ένας επεξεργαστής κάνει μία εγγραφή—π.χ. `write(A)` στο παράδειγμά μας—οι υπόλοιποι επεξεργαστές την αντιλαμβάνονται μόνο όταν διαβάσουν τη μεταβλητή—`read(A)`. Έτσι, στο παραπάνω παράδειγμα, το μοντέλο συνέπειας του συστήματος θα μπορούσε να εγγυάται ότι, εφόσον η εγγραφή στο `B` έπεται της εγγραφής του `A`, οι επεξεργαστές θα αντιληφθούν τη νέα τιμή του `B` μετά τη νέα τιμή του `A`. Αν δηλαδή το `read(B)` επιστρέψει τη νέα τιμή του `B`, θα ήταν εγγυημένο ότι το `read(A)` θα επιστρέψει τη νέα τιμή του `A`. Το μοντέλο συνέπειας του συστήματος που αγοράσαμε δεν το εγγυάται αυτό, όπως είδαμε. Δε σημαίνει ότι είναι χαλασμένο (κατά πάσα πιθανότητα!), απλά ότι η συνέπεια της μνήμης του είναι λίγο «περίεργη».

Στη συνέχεια, θα γνωρίσουμε διάφορα μοντέλα συνέπειας, θα δούμε πώς λειτουργούν και γιατί μπορεί να τα προτιμήσει κανείς έναντι άλλων.

2.7.1 Ακολουθιακή συνέπεια

Όταν εκτελούμε ένα σειριακό πρόγραμμα, είναι βέβαιο ότι οι εντολές εκτελούνται σειριακά, η μία μετά την άλλη ή *με τη σειρά του προγράμματος* (program order), όπως λέγεται. Όμως, σε ένα παράλληλο πρόγραμμα, κάθε επεξεργαστής εκτελεί διαφορετική διεργασία ασύγχρονα με τους άλλους, και, όπως καταλαβαίνουμε, δεν μπορεί να υπάρχει μία προκαθορισμένη συνολική σειρά εκτέλεσης. Η «σειρά προγράμματος», σε αυτή την περίπτωση, ορίζεται ξεχωριστά για τη διεργασία που εκτελεί ο κάθε επεξεργαστής.

Το μοντέλο συνέπειας που διαισθητικά αναμένει κάθε προγραμματιστής είναι η λεγόμενη *ακολουθιακή συνέπεια* (sequential consistency), η οποία προσδιορίζει τα εξής:

1. Οι λειτουργίες μνήμης (ανάγνωση, εγγραφή) που εκτελεί ένας οποιοσδήποτε επεξεργαστής ολοκληρώνονται με την σειρά του προγράμματος που εκτελεί·
2. όλες οι λειτουργίες μνήμης ολοκληρώνονται με κάποια σειρά μεταξύ τους.

Το πρώτο σημείο μάλλον είναι αυτονόητο. Το δεύτερο σημείο θέλει κάποια επεξήγηση. Αυτό που εννοεί είναι ότι η μνήμη, ως σύνολο, σειριοποιεί *όλες* τις προσπελάσεις που δέχεται. Μπορεί να δέχεται ταυτόχρονες εγγραφές και αναγνώσεις για διαφορετικά πιθανώς δεδομένα από πολλούς επεξεργαστές, όμως, διεκπεραιώνονται η μία μετά την άλλη, σειριακά.

Ως αποτέλεσμα, οι λειτουργίες μνήμης εμφανίζονται *ατομικές*, δηλαδή μία λειτουργία εκτελείται και ολοκληρώνεται χωρίς να παρεμβληθεί άλλη λειτουργία. Όταν λέμε ότι μία εγγραφή «ολοκληρώνεται», εννοούμε ότι όλο το σύστημα μνήμης βλέπει την εγγραφή αυτή (π.χ. όλες οι κρυφές μνήμες έχουν λάβει τη νέα τιμή, αν έχουμε πρωτόκολλο εγγραφής-ενημέρωσης ή έχουν ακυρώσει το δεδομένο, αν έχουμε πρωτόκολλο εγγραφής-ακύρωσης για τη συνοχή), ούτως ώστε, αν αμέσως μετά γίνει ανάγνωση από κάποιον επεξεργαστή, αυτός να λάβει τη νέα τιμή.

Γυρνώντας στο παράδειγμά μας (Σχ. 2.19), ένα σύστημα το οποίο διαθέτει ακολουθιακή συνέπεια, δεν μπορεί να παράγει την εκτύπωση `10` διότι καταστρατηγείται το πρώτο σημείο. Από τη στιγμή που στην Δ1 η σειρά προγράμματος λέει ότι το `B = 1` θα ολοκληρωθεί μετά το `A = 1`, δεν μπορεί καμία διεργασία να δει (`read`) πρώτα τη δεύτερη εγγραφή και μετά την πρώτη.

Η ακολουθιακή συνέπεια είναι αυτό που σιωπηρά υποθέτει κάθε προγραμματιστής ότι ισχύει στο σύστημα. Ως ένα ακόμα παράδειγμα, ας δούμε το Σχ. 2.20, το οποίο αποτελεί τμήμα του γνωστού αλγόριθμου του Dekker για αμοιβαίο αποκλεισμό. Ο αλγόριθμος δουλεύει μόνο αν το σύστημα διαθέτει ακολουθιακή συνέπεια! Αλλιώς, υπάρχει περίπτωση και οι δύο διεργασίες να εισέλθουν στην κρίσιμη περιοχή. Ας δούμε γιατί: αν δεν υπάρχει ακολουθιακή συνέπεια τότε μπορεί να γίνει το `A = 1`, να μπει η Δ1 στην κρίσιμη περιοχή, η Δ2 να μην προλάβει να δει ότι το `A` έγινε 1 και να μπει και αυτή στην κρίσιμη περιοχή. Ουσιαστικά, έχει καταστρατηγηθεί το δεύτερο σημείο της ακολουθιακής συνέπειας, λόγω της μη ατομικότητας της εγγραφής του `A`.

Αρχικά είχαμε `A = B = 0`

Δ1	**Δ2**

`A = 1;`
`if (B == 0)`
`  <critical section>`
	`B = 1;`
	`if (A == 0)`
	`  <critical section>`

Σχήμα 2.20 *Τμήμα αλγορίθμου αμοιβαίου αποκλεισμού.*

Δυστυχώς, όμως, η ακολουθιακή συνέπεια μπορεί είτε να είναι δύσκολο να υλοποιηθεί σε ένα σύστημα, είτε να θέτει περιορισμούς στην ταχύτητά του. Ένας σύγχρονος επεξεργαστής αντλεί τις επιδόσεις του από τεχνικές, όπως η επικαλυπτόμενη εκτέλεση εντολών, η εκτέλεση εκτός σειράς (out-of-order execution) και η εικαζόμενη εκτέλεση (speculative execution), οι οποίες *αλλοιώνουν τη σειρά προγράμματος*. Επομένως, οι επεξεργαστές αυτοί δεν μπορούν εύκολα να χρησιμοποιηθούν σε μηχανήματα με ακολουθιακή συνέπεια.

Υπάρχει πάντως, κάποιο περιθώριο για τη χρήση μερικών από αυτές τις τεχνικές. Για παράδειγμα, ο επεξεργαστής MIPS R10000, ο οποίος χρησιμοποιείται στο σύστημα SGI Origin 2000, μπορεί να εκτελεί ταυτόχρονα δύο και παραπάνω εντολές (π.χ. μπορεί να ξεκινήσει μία αριθμητική πράξη ή μία ανάγνωση πριν ολοκληρωθεί μία εγγραφή προς τη μνήμη) και μάλιστα, να τις *ξεκινά* εκτός σειράς προγράμματος. Όμως, όλες τις αναφορές στη μνήμη τις *ολοκληρώνει* με τη σειρά προγράμματος. Από τη στιγμή όμως, που επιτρέπει ταυτόχρονα να βρίσκονται σε εξέλιξη παραπάνω από μία εντολές - αναφορές προς τη μνήμη (άσχετα αν ολοκληρώνονται με τη σωστή σειρά), ουσιαστικά δεν διαθέτει *ατομικότητα* στις προσπελάσεις. Έτσι, από μόνος του ο επεξεργαστής δεν εξασφαλίζει την ακολουθιακή συνέπεια. Αν την ατομικότητα την εξασφαλίσει το υλικό του υποσυστήματος της μνήμης, τότε το σύστημα θα διαθέτει ακολουθιακή συνέπεια. Κάτι τέτοιο συμβαίνει και στο SGI Origin 2000.

Εκτός από τους επεξεργαστές όμως, μπαίνουν περιορισμοί και στους μεταφραστές (compilers). Βασικές τεχνικές βελτιστοποιήσεων, όπως η απαλοιφή υποεκφράσεων, η διάδοση των σταθερών και οι μετασχηματισμοί βρόχων είναι πλέον απαγορευμένες, αφού αλλάζουν τη σειρά προγράμματος. Η χρήση καταχωρητών για την αποθήκευση μεταβλητών είναι επίσης αδύνατη, γιατί συνήθως το υλικό συνοχής ασχολείται με τις λανθάνουσες μνήμες και την κύρια μνήμη και δεν έχει άμεση γνώση για το περιεχόμενο των καταχωρητών. Είναι γενικά ενδεδειγμένη η χρήση των μεταβλητών `volatile` στη γλώσσα C, αν πρόκειται για ευαίσθητες κοινόχρηστες μεταβλητές του προγράμματος (π.χ. μεταβλητές που χρησιμοποιούνται να σημάνουν κάποια συνθήκη στην οποία βασίζονται άλλες διεργα-

σίες, όπως στον αλγόριθμο του Dekker που είδαμε προηγουμένως). Οι μεταβλητές `volatile` δεν αποθηκεύονται σε καταχωρητές και η προσπέλασή τους γίνεται πάντα με τη σειρά του προγράμματος.

Οι παραπάνω περιορισμοί που θέτει η ακολουθιακή συνέπεια στο σύστημα και στους μεταφραστές δίνουν το κίνητρο για υιοθέτηση πιο «χαλαρών» (αν και λιγότερο λογικών) μοντέλων συνέπειας, τα οποία θα δούμε στη συνέχεια.

2.7.2 Χαλαρώνοντας τη συνέπεια

Ας δούμε ένα παράδειγμα, από το οποίο φαίνεται και αριθμητικά το όφελος που μπορεί να έχουμε αν δεν επιμείνουμε στην ακολουθιακή συνέπεια.

Παράδειγμα 2.7

Υποθέστε ένα σύστημα με πρωτόκολλο καταλόγων για τη συνοχή της λανθάνουσας μνήμης, το οποίο δουλεύει ως εξής. Αν μία CPU X θέλει να τροποποιήσει ένα δεδομένο, τότε ζητά από την κύρια μνήμη την «αποκλειστικότητα» και έστω ότι για αυτό χρειάζεται χρόνος ίσος με 20 κύκλους ρολογιού. Η κύρια μνήμη, συμβουλευόμενη τον κατάλογο, θα στείλει μηνύματα ακύρωσης του δεδομένου σε όσες λανθάνουσες μνήμες διαθέτουν αντίγραφο. Ας υποθέσουμε ότι κάθε μήνυμα ακύρωσης παίρνει 10 κύκλους ρολογιού και ότι τη δεδομένη χρονική στιγμή υπάρχουν άλλες 4 λανθάνουσες μνήμες που έχουν αντίγραφα. Τέλος, κάθε λανθάνουσα μνήμη χρειάζεται 50 κύκλους για να ακυρώσει το δεδομένο της και να στείλει γνωστοποίηση (acknowledgement) στην κύρια μνήμη. Αφού η κύρια μνήμη συλλέξει όλες τις γνωστοποιήσεις, στέλνει μήνυμα με παραχώρηση αποκλειστικότητας στη CPU X, απαιτώντας χρόνο 20 κύκλων. Πόσος χρόνος χρειάζεται συνολικά για την τροποποίηση;

Απάντηση:

Μετά τους 20 πρώτους κύκλους, η κύρια μνήμη στέλνει 4 ακυρώσεις. Η τελευταία θα φύγει στον κύκλο $20 + 4 \times 10$. Η λανθάνουσα μνήμη που θα το λάβει τελευταία (οι άλλες εργάζονται ταυτόχρονα), θα απαντήσει μετά από 50 κύκλους και τέλος θα χρειαστούν άλλοι 20 κύκλοι για ενημέρωση της CPU X. Σύνολο, δηλαδή, $20 + 4 \times 10 + 50 + 20 = 130$ κύκλοι.

Μία λογική ερώτηση είναι τώρα η εξής: αφού η CPU X θα πάρει ούτως ή άλλως την αποκλειστικότητα και αφού όλες οι άλλες CPU θα ενημερωθούν κάποια στιγμή, γιατί να περιμένει να ενημερωθούν όλοι και να μην την πάρει αμέσως μετά τους πρώτους 20 κύκλους; Η απάντηση είναι ότι φυσικά και μπορεί να την πάρει! Τότε, βέβαια, η εγγραφή δεν θα είναι ατομική—δηλαδή κατά την διάρκεια εκτέλεσής της μπορούν να παρεμβληθούν άλλες λειτουργίες. Ως αποτέλεσμα, μπορεί κάποια CPU να λάβει

νωρίτερα μία «πιο πρόσφατη» εγγραφή από μία «παλαιότερη», καταστρατηγώντας έτσι την ακολουθιακή συνέπεια. Όμως, η επιτάχυνση κατά 110 κύκλους της CPU X είναι σημαντική.

Γενικά, είναι αποδεκτό ότι χαλαρώνοντας τις απαιτήσεις της ακολουθιακής συνέπειας αυξάνουμε τα περιθώρια για βελτιστοποιήσεις ταχύτητας, τόσο στο υλικό όσο και στους μεταφραστές. Για τον λόγο αυτό, τα περισσότερα συστήματα υποστηρίζουν πιο χαλαρά μοντέλα συνέπειας της μνήμης. Τα μοντέλα αυτά, όπως είναι αναμενόμενο, δεν εξασφαλίζουν τη συμπεριφορά που αναμένει διαισθητικά ο προγραμματιστής από το σύστημα, αν και είναι γεγονός ότι οι εφαρμογές χρήστη σπάνια επηρεάζονται από το μοντέλο συνέπειας. Πιο ευαίσθητοι είναι οι κώδικες που υλοποιούν λειτουργίες συγχρονισμού μεταξύ των διεργασιών (π.χ. δείτε τον αλγόριθμο του Dekker για αμοιβαίο αποκλεισμό στις προηγούμενες παραγράφους), που ως επί το πλείστον, συναντώνται είτε στο λειτουργικό σύστημα είτε σε βιβλιοθήκες του συστήματος.

Τα συστήματα που δεν διαθέτουν ακολουθιακή συνέπεια συνήθως παρέχουν επιπλέον εντολές ή μηχανισμούς, ώστε ο χρήστης να μπορεί να επιβάλλει κατά κάποιο τρόπο την ακολουθιακή συνέπεια σε όποιο σημείο απαιτείται. Αυτές οι «δικλίδες ασφαλείας» είναι γνωστές ως *φραγές μνήμης* (memory barriers) ή *φράχτες* (fences).

Προκειμένου να χαλαρώσουμε την ακολουθιακή συνέπεια, θα πρέπει να χαλαρώσουμε μία ή και τις δύο από τις προϋποθέσεις της, δηλαδή την εκτέλεση σε σειρά προγράμματος για κάθε επεξεργαστή και τη συνολική σειριοποίηση των προσπελάσεων στη μνήμη—που μεταφράζεται σε ατομικότητα των προσπελάσεων. Αυτό που επιτρέπουν, ουσιαστικά, όλα τα χαλαρά μοντέλα συνέπειας είναι η εκτέλεση προσπελάσεων εκτός σειράς προγράμματος.

Ας υποθέσουμε ότι σε έναν επεξεργαστή έχουμε προσπέλαση δύο διαφορετικών μεταβλητών[1]. Οι πιθανές περιπτώσεις προσπελάσεων είναι τέσσερις:

W→R	W→W	R→W	R→R
`write(A)`	`write(A)`	`read(A)`	`read(A)`
`read(B)`	`write(B)`	`write(B)`	`read(B)`

Χρησιμοποιούμε τον συμβολισμό X→Y για να υποδηλώσουμε ότι η λειτουργία Υ έπεται της X, όπου οι λειτουργίες είναι ανάγνωση (R) ή εγγραφή (W).

[1]Να σημειώσουμε, για την περίπτωση που δεν έχει γίνει κατανοητό, ότι προσπελάσεις της ίδιας μεταβλητής δεν υπόκεινται σε αναδιάταξη. Για παράδειγμα, αν γράφουμε μία μεταβλητή και μετά τη διαβάζουμε δεν υπάρχει περίπτωση να αλλάξει η σειρά των λειτουργιών από τη CPU ή από τον μεταφραστή, καθώς υπάρχει εξάρτηση μεταξύ των δύο λειτουργιών. Η εκτέλεση εκτός σειράς προγράμματος γίνεται μόνο αν η αναδιάταξη των εντολών δεν καταστρέφει τις εξαρτήσεις, π.χ. όταν πρόκειται για προσπελάσεις διαφορετικών μεταβλητών. Απλά, η αναδιάταξη αυτή μπορεί να επηρεάσει τις λογικές εξαρτήσεις που δημιουργούνται με αυτά που εκτελούν *οι άλλες* CPU—κάτι που μόνο η ακολουθιακή συνέπεια το αποτρέπει σε όλες τις περιπτώσεις.

Επιτρέποντας αναδιάταξη της σειράς W→R. Στα μοντέλα που επιτρέπουν αυτή την αναδιάταξη, μπορεί μία επόμενη ανάγνωση να ξεκινήσει πριν ολοκληρωθεί μία προηγούμενη εγγραφή. Τη συγκεκριμένη αναδιάταξη την επιτρέπουν όλα τα μοντέλα χαλαρής συνέπειας. Είναι η μόνη αναδιάταξη που επιτρέπουν τα μοντέλα της *συνολικής σειράς αποθήκευσης* (total store order—TSO) και της *συνέπειας επεξεργαστή* (processor consistency—PRC). Το μοντέλο TSO υποστηρίζεται από τους επεξεργαστές SPARC της SUN ενώ, μεταξύ άλλων, οι επεξεργαστές Pentium Pro της Intel διέθεταν συνέπεια επεξεργαστή. Η διαφορά των δύο μοντέλων είναι ότι το μοντέλο συνέπειας επεξεργαστή χαλαρώνει και την ατομικότητα των εγγραφών. Μπορεί, πιο συγκεκριμένα, πριν ενημερωθούν όλες οι CPU για τη νέα τιμή του δεδομένου, κάποια από όλες να μπορέσει να διαβάσει τη νέα αυτή τιμή.

Τα μοντέλα αυτά, γενικά δεν προκαλούν προβλήματα στον προγραμματιστή καθώς είναι ελάχιστα τα προγράμματα που η ορθότητά τους εξαρτάται από τη διατήρηση της σειράς W→R. Π.χ. το πρώτο μας παράδειγμα (Σχ. 2.19) δουλεύει σωστά σε συστήματα TSO και PRC, καθώς οι εγγραφές των `A` και `B` δεν αναδιατάσσονται. Το ίδιο συμβαίνει και στο Σχ. 2.21, κλασικό παράδειγμα κώδικα αναμονής για κάποια συνθήκη που τη σηματοδοτεί μία άλλη διεργασία· δεν υπάρχει περίπτωση να τυπωθεί `0`, αφού οι εγγραφές των `A` και `flag` ολοκληρώνονται με τη σωστή σειρά.

Αρχικά είχαμε `A = flag = 0`

Δ1	Δ2

`A = 1;`
`flag = 1;`
	`while (flag == 0)`
	`  ;`
	`printf("%d", A);`

Σχήμα 2.21 *Συνήθης κώδικας αναμονής για κάποια συνθήκη.*

Σε αντίθεση όμως με την ακολουθιακή συνέπεια, ο κώδικας στο Σχ. 2.20 μπορεί να οδηγήσει και τις δύο διεργασίες στην κρίσιμη περιοχή, αφού οι αναγνώσεις μπορεί να ολοκληρωθούν πριν τις εγγραφές. Μία ακόμα περίπτωση δίνεται στο Πρόβλημα 2.8.

Προκειμένου να απαγορευτεί η αναδιάταξη των W→R και οι δύο εντολές να εκτελεστούν με τη σειρά που εμφανίζονται, αρκεί ανάμεσά τους να παρεμβληθεί μία εντολή τύπου ανάγνωσης-τροποποίησης-εγγραφής (read-modify-write, RMW). Οι εντολές αυτές δεν είναι ούτε καθαρά ανάγνωσης ούτε καθαρά εγγραφής, είναι και τα δύο. Άρα, μία εντολή RMW δεν αναδιατάσσεται με την εγγραφή που προηγείται, αλλά ούτε και με την ανάγνωση που έπεται. Έτσι, πριν εκτελεστεί μία τέτοια εντολή είναι σίγουρο ότι έχουν εκτελεστεί όλες οι προηγούμενες. Επίσης, η εντολή ανάγνωσης που ακολουθεί θα εκτελεστεί μετά την

ολοκλήρωση της εντολής RMW, επιβάλλοντας έτσι στο συγκεκριμένο σημείο τη σειρά προγράμματος. Στους επεξεργαστές SPARC μπορεί να χρησιμοποιηθεί και η εντολή MEMBAR, την οποία θα δούμε παρακάτω.

Επιτρέποντας την αναδιάταξη της σειράς W→R, ουσιαστικά επιτρέπονται πολλές από τις προχωρημένες τεχνικές επιτάχυνσης που συναντά κανείς στους μοντέρνους επεξεργαστές (η χρονική επικάλυψη μίας ανάγνωσης με μία εγγραφή «κρύβει» μέρος από την καθυστέρηση ολοκλήρωσης της εγγραφής). Από την άλλη μεριά, δεν αφήνει πολλά περιθώρια βελτιστοποιήσεων στους μεταφραστές, αφού πολλές από αυτές τις τεχνικές απαιτούν την ευχέρεια αναδιάταξης οποιονδήποτε δύο (ανεξάρτητων) εντολών. Τέτοιου είδους βελτιστοποιήσεις μπορούν να γίνουν αν το μοντέλο συνέπειας είναι πολύ πιο χαλαρό.

Επιτρέποντας οποιαδήποτε αναδιάταξη. Στην κατηγορία αυτή, επιτρέπεται σε κάθε CPU η εκτέλεση οποιονδήποτε (μη εξαρτώμενων πάντα) εντολών προσπέλασης μνήμης, εκτός σειράς προγράμματος. Συμπεριλαμβάνονται μοντέλα, όπως της *ασθενούς σειράς* (weak order, WO), της *συνέπειας αποδέσμευσης* (release consistency, RC), καθώς και μοντέλα που υποστηρίζονται από συγκεκριμένους εμπορικούς επεξεργαστές και τα οποία έχουν μικροδιαφορές μεταξύ τους.

Στο μοντέλο WO οι λειτουργίες μνήμης είναι δύο τύπων: προσπελάσεις δεδομένων και προσπελάσεις συγχρονισμού. Προκειμένου να διατηρηθεί η σειρά προγράμματος μεταξύ δύο προσπελάσεων, θα πρέπει ο προγραμματιστής να προσδιορίσει τη μία τουλάχιστον ως προσπέλαση συγχρονισμού. Μεταξύ δύο προσπελάσεων συγχρονισμού επιτρέπεται, φυσικά, οι προσπελάσεις δεδομένων να εκτελούνται με οποιαδήποτε σειρά.

Στο μοντέλο RC υπάρχουν δύο τύποι προσπελάσεων συγχρονισμού, η *δέσμευση* (acquire) και η *αποδέσμευση* (release). Οι συνηθισμένες προσπελάσεις δεδομένων που ακολουθούν (στη σειρά προγράμματος) μία προσπέλαση δέσμευσης μπορούν να ξεκινήσουν μόνο αν τελειώσει η προσπέλαση δέσμευσης. Οι προσπελάσεις αποδέσμευσης γίνονται, αφού πρώτα ολοκληρωθούν όλες οι εκκρεμείς (οι προηγούμενες, δηλαδή, στη σειρά προγράμματος) προσπελάσεις.

Τα μοντέλα αυτά παρέχουν πλήρη ελευθερία στους μεταφραστές για βελτιστοποιήσεις και (θεωρητικά τουλάχιστον) μπορούν να αντλήσουν αυξημένες επιδόσεις από το υλικό. Όμως, ο προγραμματιστής πρέπει πλέον να είναι προσεκτικός. Προκειμένου να εξασφαλίσει τη σειρά προγράμματος σε ευαίσθητα σημεία του κώδικα, θα πρέπει πρώτα να τα εντοπίσει (!) και στη συνέχεια να εισάγει μόνος του επιπλέον εντολές, που παρέχονται πάντα στα συστήματα αυτά. Για παράδειγμα, οι επεξεργαστές Alpha της DEC διαθέτουν δύο φράχτες: την εντολή MB που λειτουργεί όπως μία κλήση συγχρονισμού στο μοντέλο WO (περιμένει, δηλαδή, να ολοκληρωθούν όλες οι εκκρεμείς προσπελάσεις) και την εντολή WMB που επιβάλλει σειρά προγράμματος μόνο μεταξύ εγγραφών (σαν την STBAR των SPARC). Οι επεξεργαστές PowerPC της IBM διαθέτουν μόνο τον φράχτη SYNC που λειτουργεί, όπως το MB των Alpha. Οι επεξεργαστές SPARC (έκδοση V9), οι οποίοι υποστηρίζουν το λεγόμενο μο-

ντέλο *χαλαρής σειράς μνήμης* (relaxed memory order, RMO)[2], διαθέτουν μία ευέλικτη εντολή MEMBAR, η οποία διαθέτει τέσσερα bit επιλογών για να προσδιορίσει κανείς ποιες (ποιον συνδυασμό) από τις τέσσερις σειρές W→R, W→W, R→W, R→R επιθυμεί να διατηρηθούν. Με κατάλληλη χρήση του φράχτη αυτού, είναι δυνατόν να εξομοιωθεί πρακτικά οποιοδήποτε από τα μοντέλα συνέπειας έχουμε περιγράψει μέχρι τώρα.

2.7.3 Συνοψίζοντας τη συνέπεια μνήμης

Όπως είδαμε, αυτό που εμείς θεωρούμε «προφανές», δεν είναι πάντα αυτό που υποστηρίζει ένας πολυεπεξεργαστής. Η ακολουθιακή συνέπεια απελευθερώνει τον προγραμματιστή από τις λεπτομέρειες του υλικού, αλλά σίγουρα δεν αφήνει χώρο ούτε στο υλικό ούτε στους μεταφραστές για σημαντικές βελτιστοποιήσεις. Από την άλλη μεριά, τα χαλαρά μοντέλα συνέπειας απελευθερώνουν το υλικό και τους μεταφραστές από περιορισμούς, αλλά μεταθέτουν την ευθύνη της ορθής λειτουργίας του προγράμματος στον προγραμματιστή. Είναι αλήθεια ότι στην πλειονότητα των εφαρμογών, ακόμα και το πιο χαλαρό μοντέλο συνέπειας δε θα επηρεάσει την ορθή λειτουργία. Όμως, ο προγραμματιστής πρέπει να γνωρίζει τους κινδύνους.

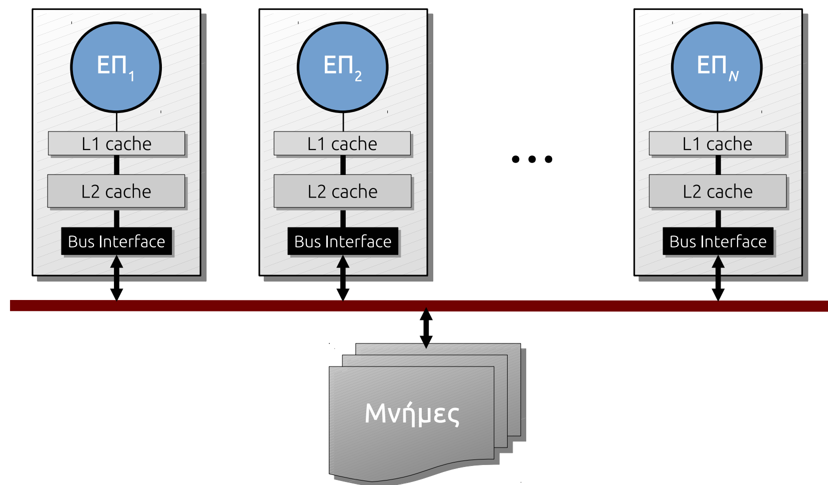
Πάντως, έχει υποστηριχθεί ότι οι διαφορές στις επιδόσεις δεν είναι τόσο μεγάλες ώστε να κάνουμε τη ζωή του προγραμματιστή πιο δύσκολη (ειδικά σε σύγχρονους επεξεργαστές που υποστηρίζουν εικαζόμενη εκτέλεση). Η ακολουθιακή συνέπεια και μοντέλα όπως το PRC, που δεν είναι ιδιαίτερα χαλαρά, μάλλον πρέπει να προτιμούνται.

2.8 Επεξεργαστές πολλαπλών πυρήνων

Μέχρι τώρα, έχουμε υποθέσει ότι οι επεξεργαστές του συστήματος είναι *μονοπύρηνοι*, αποτελούν δηλαδή ο καθένας τους ένα και μόνο επεξεργαστικό στοιχείο. Τα τελευταία χρόνια όμως, επεξεργαστές τέτοιας μορφής είναι όλο και δυσκολότερο να βρει κανείς ακόμα και στα πιο ανίσχυρα υπολογιστικά συστήματα. Οι επεξεργαστές *πολλαπλών πυρήνων* είναι η νέα πραγματικότητα, όπου κάθε επεξεργαστής εμπεριέχει δύο ή παραπάνω επεξεργαστικά στοιχεία (πυρήνες). Μεταξύ αυτών γίνεται μία αόριστη διάκριση ως προς το πλήθος των πυρήνων που περιέχουν:

- ως multicore αναφέρονται επεξεργαστές με σχετικά λίγους πυρήνες, συνήθως έως 32,

[2] Οι επεξεργαστές SPARC-V8 (32 bit) υποστηρίζουν τα μοντέλα TSO και PSO ενώ οι SPARC-V9 (64 bit) υποστηρίζουν επιπρόσθετα το μοντέλο RMO. Το ποιο μοντέλο συνέπειας θα ακολουθεί ο επεξεργαστής το καθορίζει κανείς γράφοντας μία κατάλληλη τιμή σε συγκεκριμένο καταχωρητή της CPU.



Σχήμα 2.22 *Ιεραρχία μνήμης σε ένα πολυεπεξεργαστικό σύστημα κοινόχρηστης μνήμης.*

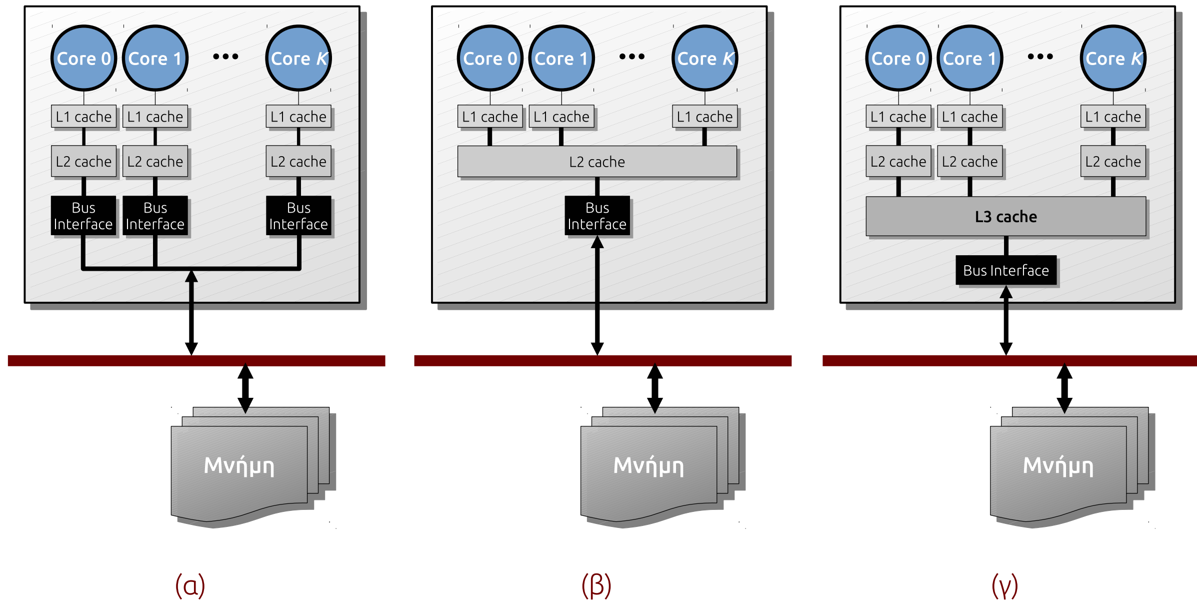
- ως manycore αναφέρονται επεξεργαστές με πολλούς πυρήνες, όπου ως πολλοί συνήθως εννοούνται 64 και παραπάνω.

Εμείς εδώ, όταν λέμε «πολυπύρηνοι» θα αναφερόμαστε σε όλους τους επεξεργαστές πολλαπλών πυρήνων. Αν χρειάζεται να μιλήσουμε ειδικά για τη δεύτερη κατηγορία, θα αναφερόμαστε σε επεξεργαστές *πάρα πολλών* πυρήνων. Τέλος, με τον όρο «ολιγοπύρηνοι» θα εννοούμε επεξεργαστές με σχετικά μικρό αριθμό πυρήνων (2 – 8 περίπου).

Η πρώτη προσέγγιση που έχει κανείς, είναι ότι ένα σύστημα που διαθέτει έναν πολυπύρηνο επεξεργαστή με N πυρήνες, σε μεγάλο βαθμό είναι ισοδύναμο με ένα σύστημα με N μονοπύρηνους επεξεργαστές. Και είναι όντως θεμιτή αυτή η προσέγγιση, αν και υπάρχουν αρκετές και σημαντικές διαφορές, όπως θα δούμε. Ό,τι έχουμε πει μέχρι τώρα, λοιπόν, για τους πολυεπεξεργαστές ισχύει και για συστήματα με πολυπύρηνους επεξεργαστές, απλά θεωρώντας τον κάθε πυρήνα ως αυτόνομο, μονοπύρηνο επεξεργαστή.

Η παρουσία, όμως, πολυπύρηνων επεξεργαστών επιβάλλει αλλαγές στην οργάνωση των συστημάτων κοινόχρηστης μνήμης, καθώς υπεισέρχονται πολυπλοκότερες σχέσεις και διασυνδέσεις μεταξύ επεξεργαστών και μνήμης. Η κλασική ιεραρχία της μνήμης που φαίνεται στο Σχ. 2.22 περιλαμβάνει, για κάθε επεξεργαστή, ένα ή δύο επίπεδα κρυφής μνήμης (L1 cache και L2 cache) και ως τρίτο επίπεδο στην ιεραρχία θεωρούνται οι κοινόχρηστες μνήμες τις οποίες μοιράζονται οι επεξεργαστές μέσω του δικτύου που ενώνει επεξεργαστές και μνήμες.

Η τοποθέτηση πολλαπλών πυρήνων μέσα σε έναν πολυπύρηνο επεξεργαστή μπορεί να γίνει με αρκετούς διαφορετικούς τρόπους. Η απλούστερη οργάνωση φαίνεται στο Σχ. 2.23(α). Αν προσέξετε, πρόκειται για απλή τοποθέτηση K επεξεργαστών από το Σχ. 2.22 μέσα σε ένα ολοκληρωμένο κύκλωμα, με μοναδική διαφορά το γεγονός ότι αντί να έχουμε K συνδέσεις προς το δίκτυο, όλοι οι πυρήνες πολυπλέκουν τις συνδέσεις τους επάνω σε



Σχήμα 2.23 *Ιεραρχίες μνήμης σε επεξεργαστές πολλαπλών πυρήνων.*

ένα κανάλι. Η προσέγγιση αυτή απαιτεί τις λιγότερες τροποποιήσεις στην αρχιτεκτονική ενός επεξεργαστή και για το λόγο αυτό, πολλές από τις πρώιμες γενιές των πολυπύρηνων επεξεργαστών την ακολούθησαν, όπως για παράδειγμα οι επεξεργαστές Pentium D της Intel.

Είναι φανερό, όμως, ότι η οργάνωση αυτή έχει κάποια μειονεκτήματα, όπως για παράδειγμα, ότι για να επικοινωνήσουν οι πυρήνες μεταξύ τους θα πρέπει να το κάνουν μέσω των συνδέσεων προς τη μνήμη, περνώντας όλη την ιεραρχία των κρυφών μνημών. Αυτό το πρόβλημα το λύνει η οργάνωση του Σχ. 2.23(β). Οι πυρήνες πλέον, μοιράζονται την κρυφή μνήμη δευτέρου επιπέδου, καθώς και τα κυκλώματα σύνδεσης με το δίκτυο. Όχι μόνο μπορούν να επικοινωνούν αμεσότερα και ταχύτερα μέσω της κοινής κρυφής μνήμης, αλλά επιτυγχάνεται και οικονομία κλίμακας, αφού δεν χρειάζεται να δαπανάται χώρος στο ολοκληρωμένο κύκλωμα για πολλαπλές κρυφές μνήμες και για πολλαπλά κανάλια προς το δίκτυο, ελευθερώνοντας πολύτιμο πυρίτιο για επιπλέον λειτουργικότητα ή παραπάνω πυρήνες. Η γενιά Core2 Duo των επεξεργαστών της Intel στηρίχθηκε σε αυτή την αρχιτεκτονική, με κρυφή μνήμη δεύτερου επιπέδου μεγέθους 2-4MiB.

Η ύπαρξη μόνο δύο επιπέδων κρυφής μνήμης και μάλιστα με το δεύτερο επίπεδο διαμοιραζόμενο μεταξύ των πυρήνων του ίδιου επεξεργαστή, αποδεικνύεται ότι δεν έχει πάντα την αναμενόμενη απόδοση. Αυτό γίνεται εντονότερο όσο αυξάνει ο αριθμός των πυρήνων στον ίδιο επεξεργαστή. Κρίνεται επομένως, απαραίτητη η διατήρηση ιδιωτικών κρυφών μνημών επιπέδου 2 και επιπλέον η προσθήκη ενός τρίτου, κοινόχρηστου επιπέδου κρυφής μνήμης, όπως στο Σχ. 2.23(γ). Οι τετραπύρηνοι επεξεργαστές Nehalem της Intel έχουν ακριβώς αυτή την οργάνωση, με 256KiB κρυφή μνήμη επιπέδου 2 ανά πυρήνα και

κοινόχρηστη κρυφή μνήμη επιπέδου 3 που κυμαίνεται από 4 έως 18MiB, ανάλογα με τον επεξεργαστή. Από την άλλη, οι οκταπύρηνοι επεξεργαστές Bulldozer της AMD μοιράζονται ανά δύο μία κρυφή μνήμη δευτέρου επιπέδου, μεγέθους 2MiB, ενώ υπάρχει και μία κοινόχρηστη κρυφή μνήμη τρίτου επιπέδου μεγέθους 8MiB ανάμεσα στους 8 πυρήνες.

Πρέπει να είναι φανερό ότι ειδικά οι δύο τελευταίες οργανώσεις, οι οποίες διαθέτουν κοινόχρηστες κρυφές μνήμες, έχουν σημαντικά πλεονεκτήματα. Συγκεκριμένα:

- Υπάρχει ταχύτερη επικοινωνία μεταξύ των πυρήνων που μοιράζονται τις κοινόχρηστες κρυφές μνήμες, σε σύγκριση με την οργάνωση των μονοπύρηνων πολυεπεξεργαστών (Σχ. 2.22) όπου η επικοινωνία γίνεται μέσω διαύλου και κύριας μνήμης,
- μειώνεται έτσι η συνολική κίνηση στο δίαυλο του συστήματος, αφού πολλές αιτήσεις ολοκληρώνονται μέσα στον κάθε επεξεργαστή,
- πυρήνες που εργάζονται σε επικαλυπτόμενες περιοχές δεδομένων ευεργετούνται, καθώς ένας πυρήνας μπορεί να φέρνει στην κοινή κρυφή μνήμη δεδομένα που πολύ πιθανώς θα χρειαστεί και κάποιος άλλος,
- αντί ο σχεδιαστής του συστήματος να προκαθορίζει συγκεκριμένη ποσότητα ιδιωτικής κρυφής μνήμης για κάθε πυρήνα, η κοινόχρηστη κρυφή μνήμη διαμοιράζει *δυναμικά* το χώρο της. Αν ένας πυρήνας χρειάζεται λιγότερο χώρο, κάποιος άλλος μπορεί να χρησιμοποιήσει περισσότερο,
- δεν υπάρχει θέμα συνοχής σε μία κοινόχρηστη κρυφή μνήμη.

Παρ' όλα αυτά, η ύπαρξη διαμοιραζόμενης κρυφής μνήμης έχει και σημαντικά μειονεκτήματα. Μερικά από αυτά είναι τα εξής:

- Τα μεγέθη των κοινόχρηστων κρυφών μνημών που βρίσκουμε στους πολυπύρηνους επεξεργαστές είναι εν γένει μεγάλα (ξεπερνούμε σε κάποια μοντέλα τα 20MiB). Αυτός είναι και ένας από τους λόγους που οι συγκεκριμένες κρυφές μνήμες είναι σχετικά αργές,
- κρυφές μνήμες ανωτέρων επιπέδων προσπελαύνονται ταυτόχρονα από πολλαπλές κρυφές μνήμες κατωτέρων επιπέδων, κάτι που ανεβάζει σχεδιαστικά τις απαιτήσεις για μεγαλύτερο ρυθμό μεταφοράς. Επίσης, η ταυτόχρονη προσπέλαση από πολλαπλές κατώτερες κρυφές μνήμες απαιτεί κάποιου είδους διαχείριση, πράγμα το οποίο σημαίνει επιπλέον υλικό (πολλές φορές χρησιμοποιείται διασταυρωτικός διακόπτης για τις συνδέσεις) και επιπλέον καθυστερήσεις προσπέλασης,
- το πλεονέκτημα της δυναμικής διαμοίρασης του χώρου της κοινόχρηστης κρυφής

μνήμης μπορεί να γίνει και μειονέκτημα, εάν βρεθεί ένας πυρήνας ο οποίος λειτουργεί επιβαρυντικά για τους άλλους, γεμίζοντας την κρυφή μνήμη με πολλά δεδομένα που τα χρειάζεται μόνο αυτός.

Οι πολυεπεξεργαστές κοινόχρηστης μνήμης χαρακτηρίζονται και ως συστήματα *ομοιόμορφης προσπέλασης μνήμης* (Uniform Memory Access—UMA) υπό την έννοια ότι χρειάζεται ο ίδιος χρόνος προκειμένου να προσπελάσουν οποιοδήποτε δεδομένο, ανεξάρτητα από τη θέση που βρίσκεται στη μνήμη. Από την παραπάνω συζήτηση, πρέπει να συνειδητοποιήσει κανείς ότι με τις βαθύτερες και πολυπλοκότερες ιεραρχίες μνήμης που διαθέτουν, τα πολυπύρηνα συστήματα κοινόχρηστης μνήμης απέχουν αρκετά από το να χαρακτηριστούν καθαρά συστήματα UMA.

Οι επεξεργαστές πολλαπλών πυρήνων που περιγράψαμε παραπάνω είναι σχεδόν πάντα ολιγοπύρηνοι. Όταν ο αριθμός των πυρήνων αυξάνει και προχωράει προς τους πάρα πολλούς, η οργάνωση παύει να είναι αυτή της κοινόχρηστης μνήμης. Στο επόμενο κεφάλαιο θα έχουμε την ευκαιρία να δούμε πώς διαφοροποιείται τότε η αρχιτεκτονική τους.

2.9 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις

Στο κεφάλαιο αυτό γνωρίσαμε με αρκετή λεπτομέρεια την αρχιτεκτονική δομή των πολυεπεξεργαστών κοινής μνήμης. Πρόκειται για συστήματα MIMD στα οποία οι επεξεργαστές δεν έχουν ιδιωτικές μνήμες. Αντίθετα, μοιράζονται μία συλλογή από όμοιες μνήμες, η προσπέλαση των οποίων απαιτεί τον ίδιο χρόνο από όποιον επεξεργαστή και να γίνει. Γι' αυτό και τα συστήματα αυτά είναι επίσης γνωστά και ως συστήματα ομοιόμορφης προσπέλασης μνήμης (UMA—Uniform Memory Access).

Στην απλούστερη και πιο διαδεδομένη μορφή τους, τα συστήματα που μελετήσαμε βασίζονται σε κάποιο δίαυλο ο οποίος συνδέει όλους τους επεξεργαστές και όλες τις μνήμες. Είναι δε γνωστά, ως συμμετρικοί πολυεπεξεργαστές. Ο δίαυλος αποτελεί μία ελκυστική λύση για σχετικά μικρό αριθμό επεξεργαστών. Γρήγορα, όμως, φτάνει σε κορεσμό, καθώς δεν διαθέτει την χωρητικότητα και το εύρος ζώνης που απαιτείται για να εξυπηρετήσει ταυτόχρονα αιτήσεις από πολλούς επεξεργαστές. Γι' αυτό τα συστήματα αυτά ονομάζονται και *ανίκανα κλιμάκωσης* (non-scalable).

Στην περίπτωση που ο αριθμός των επεξεργαστών είναι μεγάλος, η διασύνδεση με τις μνήμες δεν πρέπει να βασίζεται σε ένα κοινό μέσο. Αντίθετα, εφαρμόζονται οι λύσεις των δικτύων με διακόπτες, όπου παρέχονται ξεχωριστές διαδρομές από κάθε επεξεργαστή σε κάθε μνήμη. Το πιο ισχυρό από αυτά τα δίκτυα είναι ο διακόπτης crossbar, ο οποίος όμως γίνεται ιδιαίτερα ακριβός και δύσκολος στην υλοποίησή του όταν αυξάνει ο αριθμός των επεξεργαστών ή / και των μνημών. Έναν συμβιβασμό μεταξύ ταχύτητας και κόστους παρέχουν τα πολυεπίπεδα δίκτυα από μικρότερους διακόπτες crossbar, όπως το Δέλτα,

το Ωμέγα, το baseline και άλλα. Για τον αναγνώστη που ενδιαφέρεται για τη σχεδίαση των δικτύων αυτών, και θέλει να εμβαθύνει στη μαθηματική θεωρία επάνω στην οποία στηρίζονται, προτείνεται το βιβλίο του A. Pattavina [Patt98].

Ανεξάρτητα από το είδος της διασύνδεσης, η γνωστή τεχνική της κρυφής μνήμης κάνει πολύ φυσιολογικά την εμφάνισή της, προκειμένου να επιταχυνθούν οι προσπελάσεις στα δεδομένα. Η χρήση, όμως, κρυφής μνήμης σε κάθε επεξεργαστή δημιουργεί προβλήματα συνοχής: ένα δεδομένο που έχει αντιγραφεί σε πολλές κρυφές μνήμες, μπορεί να έχει τροποποιηθεί ανεξάρτητα σε κάθε μία από αυτές και, μάλιστα, χωρίς να το γνωρίζει η κύρια μνήμη.

Για το σημαντικό πρόβλημα της συνοχής υπάρχουν πολλές διαφορετικές λύσεις, οι οποίες κατηγοριοποιούνται σε δύο είδη: τα πρωτόκολλα παρακολούθησης και τα πρωτόκολλα καταλόγων. Τα πρώτα απαιτούν την ύπαρξη ενός κοινού μέσου που μπορούν να το παρακολουθούν όλοι οι επεξεργαστές. Είναι, επομένως, φυσιολογικό να εφαρμόζονται σε συστήματα με διασύνδεση διαύλου. Τα πρωτόκολλα παρακολούθησης χωρίζονται περαιτέρω σε πρωτόκολλα εγγραφής-ακύρωσης και εγγραφής-ενημέρωσης, με τα πρώτα να εφαρμόζονται συχνότερα στη πράξη.

Αντίθετα, τα πρωτόκολλα καταλόγων είναι η μόνη λύση στην περίπτωση που δεν υπάρχει ο κοινός δίαυλος. Ανάλογα με την οργάνωση των πληροφοριών για τα περιεχόμενα των κρυφών μνημών, οι κατάλογοι μπορεί να είναι πλήρεις, περιορισμένοι ή αλυσιδωτοί. Τέτοια πρωτόκολλα συνοχής θα τα θυμηθούμε πάλι στο επόμενο κεφάλαιο, όταν μιλήσουμε για την λεγόμενη οργάνωση κατανεμημένης κοινής μνήμης.

Η συνέπεια του υποσυστήματος μνήμης καθορίζει τη σειρά των προσπελάσεων στη μνήμη και επηρεάζει τον τρόπο με τον οποίο αντιλαμβάνεται ένα επεξεργαστής τις προσπελάσεις που κάνουν οι υπόλοιποι. Όσο πιο χαλαρό είναι το μοντέλο συνέπειας, τόσο περισσότερα περιθώρια βελτιστοποιήσεων υπάρχουν κατά την εκτέλεση μίας εφαρμογής, αλλά και τόσο μεγαλύτερη πρέπει να είναι η προσοχή του προγραμματιστή, ώστε να αποφεύγει ανεπιθύμητες καταστάσεις.

Τα πολυπύρηνα συστήματα με μικρό αριθμό πυρήνων συμπεριφέρονται βασικά ως πολυεπεξεργαστές κοινόχρηστης μνήμης, τοποθετημένοι εξολοκλήρου σε ένα ολοκληρωμένο κύκλωμα. Παρ' όλα αυτά, η αρχιτεκτονική τους εισάγει βαθιές και σύνθετες ιεραρχίες μνήμης, οι οποίες διαφοροποιούν αρκετά τόσο την οργάνωσή τους όσο και τις επιδόσεις τους σε σχέση με τα μονοπύρηνα πολυεπεξεργαστικά συστήματα [BDM09]. Στο επόμενο κεφάλαιο θα έχουμε την ευκαιρία να δούμε επιπλέον λεπτομέρειες στην οργάνωση των πολυπύρηνων επεξεργαστών.

Τα περισσότερα από τα θέματα με τα οποία ασχοληθήκαμε στο κεφάλαιο αυτό (αλλά και πολλά από αυτά που θα δούμε στο επόμενο) καλύπτονται στο βιβλίο των Culler, Singh και Gupta [CSG99], το οποίο, αν και λίγο παλαιότερο, αποτελεί ένα από τα σημαντικότερα βιβλία που έχουν γραφτεί γενικότερα για την αρχιτεκτονική των παράλληλων υπολογιστών.

Η θεωρία της συνέπειας μνήμης ξεκίνησε με την ιστορική εργασία του L. Lamport [Lamp79] στην οποία ορίστηκε αυστηρά η ακολουθιακή συνέπεια. Για τον αναγνώστη που θέλει να ασχοληθεί, μία σύντομη επισκόπηση των διαφόρων μοντέλων συνέπειας, τα οποία ακολουθούνται μέχρι σήμερα, δίνεται στην εργασία [Hill98]. Για μεγαλύτερη εμβάθυνση στο θέμα, προτείνεται το βιβλίο των Sorin, Hill και Wood [SHW11]. Το ίδιο βιβλίο είναι μία πολύ καλή πηγή για το πρόβλημα συνοχής της κρυφής μνήμης, καλύπτοντας τόσο πρωτόκολλα παρακολούθησης, όσο και καταλόγων.

Πολλές πληροφορίες για τις οικογένειες πολυπύρηνων επεξεργαστών των εταιρειών Intel και AMD μπορεί κανείς να βρει στα εγχειρίδια [Inte15a, Inte15b, AMD13]. Μεταξύ άλλων, περιγράφονται η οργάνωσή των πυρήνων, η ιεραρχία του υποσυστήματος μνήμης κι η στρατηγική της κάθε οικογένειας για τη συνοχή της κρυφής μνήμης.

Προβλήματα

2.1 – Ο διασταυρωτικός διακόπτης crossbar δείχνει εξαιρετικά απλός στο Σχ. 2.3 και όμως, το αντίθετο συμβαίνει στην πράξη. Μπορείτε να φανταστείτε πώς περίπου υλοποιείται μία από τις κάθετες γραμμές του διακόπτη;

2.2 – Σχεδιάστε πλήρως τα δίκτυα baseline 8×8 και 16×16.

2.3 – Αν αντί για υστεροεγγραφή χρησιμοποιούσαμε πολιτική διεγγραφής στις κρυφές μνήμες, θα λυνόταν το πρόβλημα της συνοχής στο παράδειγμα της Ενότητας 2.4 (με τους δύο επεξεργαστές A και B και το δεδομένο X);

2.4 – Στα διαγράμματα καταστάσεων που είδαμε στα δύο παραδείγματα της Ενότητας 2.5 δεν έχει ληφθεί υπ' όψιν ένα τρίτο είδος «ερεθίσματος» που είναι δυνατόν να προκαλέσει κάποιες ενέργειες: την *αντικατάσταση* του δεδομένου στην κρυφή μνήμη από κάποιο άλλο. Μπορείτε να σκεφτείτε τι προσθήκες πρέπει να γίνουν στο Σχ. 2.14 για να καλυφθεί και αυτή η περίπτωση;

2.5 – Το πρωτόκολλο MESI που είδαμε στο τέλος της Ενότητας 2.5, έχει τη δυνατότητα να μειώσει την κίνηση στον δίαυλο τόσο για τα σειριακά προγράμματα που μπορεί να εκτελεί ένας πολυεπεξεργαστής, όσο και για τα τμήματα των παράλληλων προγραμμάτων που χρησιμοποιούν μόνο ιδιωτικές (μη κοινόχρηστες) μεταβλητές. Μπορείτε να σκεφτείτε γιατί;

2.6 – Μπορείτε να υπολογίσετε πόσο επιπλέον αποθηκευτικό χώρο απαιτεί η χρήση κα-

ταλόγων σε ένα σύστημα με 64, 256 ή 1024 επεξεργαστές; Υπολογίστε τον ως ποσοστό της κύριας μνήμης, υποθέτοντας ότι οι κρυφές μνήμες λειτουργούν με γραμμές / μπλοκ των 64 bytes.

2.7 – Επαναλάβετε την προηγούμενη άσκηση για περιορισμένους καταλόγους. Υποθέστε ότι $K = 10$ θέσεις.

Αρχικά είχαμε `A = B = 0`

Δ1	**Δ2**	**Δ3**

`A = 1;`	`while (A == 0)`
	`  ;`
	`B = 1;`
		`while (B == 0)`
		`  ;`
		`printf("%d", A);`

Σχήμα 2.24 *Αλυσιδωτή αναμονή για κάποια συνθήκη*

2.8 – Στο Σχ. 2.24, δίνεται κώδικας αλυσιδωτής αναμονής τριών διεργασιών, στην οποία ένα σύστημα με ακολουθιακή συνέπεια θα τυπώσει 1. Μπορείτε να σκεφτείτε τι εκτύπωση μπορεί δώσει ένα σύστημα με

- συνέπεια TSO;
- συνέπεια επεξεργαστή (PRC);

3 Οργάνωση Κατανεμημένης Μνήμης

Στο κεφάλαιο αυτό, θα δούμε τα συστήματα κατανεμημένης μνήμης τα οποία μερικές φορές ονομάζονται και πολυϋπολογιστές, αφού ο συνδυασμός επεξεργαστή και ιδιωτικής μνήμης μπορεί να θεωρηθεί ως ένας μικρός αυτόνομος υπολογιστής. Η οργάνωση των υπολογιστών αυτών βασίζεται σχεδόν αποκλειστικά στο δίκτυο διασύνδεσης μεταξύ των επεξεργαστών, το οποίο θα πρέπει να είναι γρήγορο, έτσι ώστε να εισάγονται όσο το δυνατόν λιγότερες καθυστερήσεις κατά τις επικοινωνίες. Θα δούμε ποιες είναι οι επιθυμητές ιδιότητες των δικτύων διασύνδεσης και θα έχουμε την ευκαιρία να γνωρίσουμε πολλά δημοφιλή από αυτά. Θα γνωρίσουμε επίσης, τους ομαδοποιημένους πολυεπεξεργαστές, οι οποίοι αποτελούν ένα είδος διασταύρωσης των οργανώσεων κοινόχρηστης και κατανεμημένης μνήμης, μαζί με ορισμένες λεπτομέρειες που αφορούν στη λεγόμενη οργάνωση κατανεμημένης κοινής μνήμης. Θα δούμε, τέλος, πώς όλα τα παραπάνω θέματα εμφανίζονται και σε σύγχρονους επεξεργαστές πολλαπλών πυρήνων.

Στο κεφάλαιο αυτό θα ασχοληθούμε με τις μηχανές MIMD κατανεμημένης μνήμης. Τα συστήματα αυτά έχουν μία διαφορετική φιλοσοφία από εκείνα του προηγούμενου κεφαλαίου. Συγκεκριμένα, δεν υπάρχει κοινόχρηστη μνήμη μεταξύ των επεξεργαστών αλλά, αντίθετα, υπάρχουν ιδιωτικές μνήμες και ένα δίκτυο που συνδέει τους επεξεργαστές. Έτσι, ο μόνος τρόπος να προσπελαστούν από κάποιον τα δεδομένα σε μία απομακρυσμένη ιδιωτική μνήμη, είναι μέσω ανταλλαγής μηνυμάτων επάνω στις γραμμές του δικτύου. Σημειώνεται ότι αυτό έχει άμεσο αντίκτυπο και στον τρόπο που προγραμματίζεται μια τέτοια μηχανή.

Στην Ενότητα 3.1, θα γνωρίσουμε τη βασική οργάνωση των πολυεπεξεργαστών κατανεμημένης μνήμης. Θα έχουμε, επίσης, την ευκαιρία να δούμε τη λογική πίσω από το σχεδιασμό των διαδρομητών. Οι διαδρομητές είναι οι μονάδες εκείνες οι οποίες διεκπεραιώνουν τις επικοινωνίες στο δίκτυο διασύνδεσης των επεξεργαστών.

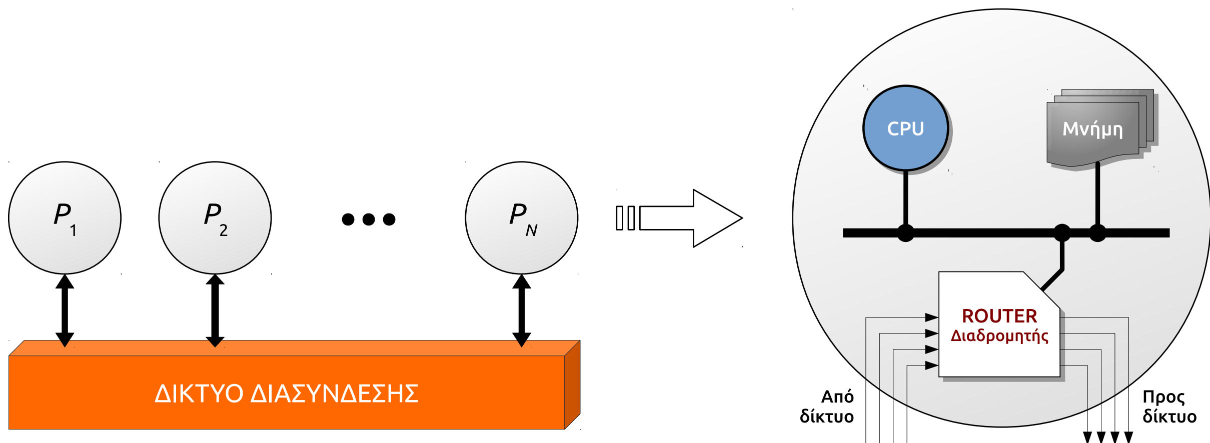
Η τοπολογία του δικτύου θα μας απασχολήσει στην Ενότητα 3.2. Χρησιμοποιώντας γράφους θα παρουσιάσουμε, θα αναλύσουμε και θα συγκρίνουμε τα χαρακτηριστικά αρκετών δημοφιλών τοπολογιών. Εκτός όμως από την τοπολογία, ένα δίκτυο διασύνδεσης χαρακτηρίζεται και από άλλες ιδιότητες. Θα δούμε αναλυτικά τη διαδρόμηση και τη μεταγωγή στις Ενότητες 3.4 και 3.5.

Στην Ενότητα 3.6, θα αναφερθούμε στα ιδιαίτερα χαρακτηριστικά των υπολογιστικών συστάδων και θα δούμε, επίσης, πώς η αρχιτεκτονική των συστημάτων κατανεμημένης μνήμης μπορεί να συνδυαστεί με αυτή της κοινόχρηστης μνήμης. Προχωρώντας ένα βήμα παραπέρα, θα δούμε ότι, αν και η μνήμη είναι συνολικά κατανεμημένη, μπορεί να υπάρξει υποστήριξη κοινού χώρου διευθύνσεων μεταξύ των επεξεργαστών σε επίπεδο υλικού. Μπορεί, δηλαδή, ένα τέτοιο σύστημα να συμπεριφερθεί σαν μηχάνημα όπου οι μνήμες φαίνονται ως κοινές σε όλους τους επεξεργαστές. Τα συστήματα αυτά είναι γνωστά ως μηχανές κατανεμημένης κοινής μνήμης. Οι προσθήκες που απαιτούνται για την υλοποίησή τους εξετάζονται στην Ενότητα 3.7. Τέλος, στην Ενότητα 3.8, θα δούμε πώς όλα τα παραπάνω ζητήματα εμφανίζονται στην οργάνωση των σύγχρονων πολυπύρηνων επεξεργαστών.

3.1 Βασική οργάνωση

Οι πολυεπεξεργαστές κατανεμημένης μνήμης, όπως ήδη γνωρίζουμε από το εισαγωγικό κεφάλαιο, αποτελούνται από ένα σύνολο ανεξάρτητων επεξεργαστών, όπου ο καθένας διαθέτει τη δική του, ιδιωτική μνήμη. Δεν υπάρχει κοινόχρηστη μνήμη μεταξύ των επεξεργαστών, οπότε η επικοινωνία τους βασίζεται σε ένα δίκτυο που τους συνδέει, γνωστό απλά ως *δίκτυο διασύνδεσης* (interconnection network). Η διάταξη φαίνεται στο Σχ. 3.1.

Ένας επεξεργαστής μαζί με την ιδιωτική του μνήμη και τις υπόλοιπες ιδιωτικές του μονάδες, ονομάζεται *επεξεργαστικός κόμβος* ή απλά *κόμβος* του συστήματος. Στο Σχ. 3.1 φαίνεται, επίσης, η χονδρική δομή ενός κόμβου. Εκτός από τον επεξεργαστή και την τοπική



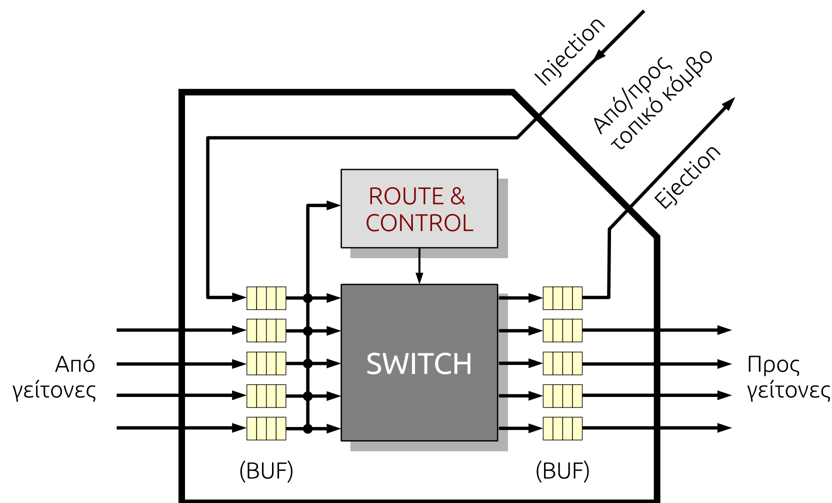
Σχήμα 3.1 *Πολυεπεξεργαστής κατανεμημένης μνήμης.*

μνήμη, υπάρχει και η μονάδα του *διαδρομητή* (router), ο οποίος είναι υπεύθυνος για την επικοινωνία του κόμβου με το δίκτυο διασύνδεσης.

Το δίκτυο διασύνδεσης θα μπορούσε να είναι οποιοδήποτε από τα δίκτυα που είδαμε στο προηγούμενο κεφάλαιο, δηλαδή δίαυλος, διακόπτης crossbar ή κάποιο πολυεπίπεδο δίκτυο. Για παράδειγμα, ο Meiko CS-2 είναι ένα μηχάνημα κατανεμημένης μνήμης που στηρίζεται σε πολυεπίπεδο διακοπτικό δίκτυο φαρδιού δέντρου (fat tree). Αυτές οι περιπτώσεις, όμως, αποτελούν εξαίρεση. Η πιο συνηθισμένη μορφή του δικτύου είναι η λεγόμενη άμεση (direct) ή σημείο-προς-σημείο (point-to-point), και αυτή είναι που θα υποθέσουμε από εδώ και στο εξής. Σε αυτά τα δίκτυα υπάρχουν ιδιωτικές συνδέσεις μεταξύ συγκεκριμένων κόμβων. Δύο κόμβοι που συνδέονται μεταξύ τους μπορούν απευθείας να επικοινωνούν και να ανταλλάσσουν μηνύματα. Η επικοινωνία μεταξύ δύο ασύνδετων επεξεργαστών γίνεται με μεταφορά μηνυμάτων μέσα από ενδιάμεσους κόμβους. Η διάταξη των επεξεργαστών στον χώρο και οι συνδέσεις μεταξύ τους αποτελούν την *τοπολογία* του δικτύου.

Με κατάλληλη σχεδίαση του δικτύου διασύνδεσης, τα συστήματα αυτά μπορούν να μεγαλώσουν πολύ σε μέγεθος με την προσθήκη περισσοτέρων κόμβων, χωρίς να έχουν προβλήματα κορεσμού, όπως έχει π.χ. ο δίαυλος στα συστήματα κοινόχρηστης μνήμης. Είναι χαρακτηριστικό ότι υπολογιστές κατανεμημένης μνήμης με δεκάδες χιλιάδες κόμβους, είχαν υλοποιηθεί από τα μέσα της δεκαετίας του 1980. Για τον λόγο αυτόν, οι αρχιτεκτονικές αυτές είναι γνωστές και ως *μαζικά παράλληλες* (massively parallel processors, MPP).

Στο Σχ. 3.2, φαίνεται η δομή ενός τυπικού διαδρομητή. Υπάρχουν κανάλια από και προς όλους τους κόμβους με τους οποίους υπάρχει απευθείας σύνδεση («γειτονικοί» κόμβοι) καθώς επίσης και από και προς τον τοπικό επεξεργαστή / μνήμη. Τα κανάλια αυτά επικοινωνούν μέσω ενός μικρού διακόπτη crossbar τον οποίο ελέγχει ένας ελεγκτής («Route & Control» στο σχήμα). Ο σκοπός του διακόπτη είναι η μεταφορά των μηνυμάτων από ένα κανάλι σε ένα άλλο. Αν, για παράδειγμα, ο δυτικός γείτονας στείλει ένα μήνυμα για τον το-



Σχήμα 3.2 *Τυπική δομή διαδρομητή.*

πικό επεξεργαστή, ο ελεγκτής θα κάνει το διακόπτη να συνδέσει το δυτικό κανάλι εισόδου με το κανάλι εξόδου που οδηγεί προς τον τοπικό κόμβο (κανάλι «Ejection»). Ταυτόχρονα, ο διαδρομητής παίζει το ρόλο του «ενδιάμεσου», προωθώντας μηνύματα μεταξύ των γειτονικών του κόμβων (π.χ. για ένα εισερχόμενο μήνυμα από τον βόρειο γείτονα που ταξιδεύει προς τον ανατολικό γείτονα, ο ελεγκτής θα δημιουργήσει την κατάλληλη σύνδεση στον crossbar προκειμένου να γίνει η μετάδοση). Τέλος, στα κανάλια εισόδου είτε / και στα κανάλια εξόδου υπάρχει συνήθως και μικρός αποθηκευτικός χώρος (buffers—«BUF» στο Σχ. 3.2) για προσωρινή αποθήκευση των μηνυμάτων, πριν την προώθησή τους. Αυτό είναι απαραίτητο στην περίπτωση που ο διακόπτης ή το επιλεγμένο κανάλι εξόδου είναι απασχολημένο με την προώθηση άλλων μηνυμάτων. Στο σχήμα, οι buffers σημειώνονται σε παρένθεση, ώστε να υποδηλώσουν ότι πιθανόν να μην υπάρχουν, ανάλογα με την αρχιτεκτονική του διαδρομητή.

Αν παρομοιάσουμε τον διαδρομητή με μία κάρτα δικτύου, ο κάθε κόμβος (Σχ. 3.1) μοιάζει με ολοκληρωμένο, αυτόνομο υπολογιστή και το όλο σύστημα θυμίζει δίκτυο υπολογιστών συνδεδεμένων μεταξύ τους μέσω των καρτών δικτύου. Αυτός είναι και ο λόγος που, μερικές φορές, χρησιμοποιείται ο όρος *πολυϋπολογιστής* (multicomputer) για αυτά τα συστήματα. Πράγματι, υπάρχουν αρκετά κοινά σημεία με τα δίκτυα υπολογιστών, αλλά και πολλές και σημαντικές διαφορές οι οποίες προκύπτουν από τα διαφορετικά προβλήματα που πρέπει να λυθούν, τόσο στα κυκλώματα σύνδεσης με το δίκτυο (δηλαδή στους διαδρομητές) όσο και στο ίδιο το δίκτυο. Στην περίπτωση, πάντως, που το σύστημα είναι όντως βασισμένο σε ανεξάρτητους υπολογιστές ενωμένους μέσω ενός τοπικού δικτύου δεδομένων, τότε μιλάμε για υπολογιστική συστάδα (cluster), κάτι που θα αναλύσουμε σε ξεχωριστή ενότητα (3.6.1).

Οι διαδρομητές θα πρέπει να πληρούν δύο προϋποθέσεις: *ταχύτητα* και *απλότητα*. Η ταχύτητα είναι απαραίτητη γιατί, σε αντίθεση με τα δίκτυα υπολογιστών, η επικοινω-

νία μεταξύ των επεξεργαστών είναι πολύ συχνή και τα δεδομένα που ανταλλάσσονται μπορεί να είναι πολύ μικρά σε μέγεθος. Επομένως, η προσπέλαση μιας μνήμης σε ένα απομακρυσμένο σημείο θα πρέπει να γίνεται όσο το δυνατόν γρηγορότερα. Οι πολύπλοκοι επικοινωνιακοί αλγόριθμοι που εφαρμόζονται στα δίκτυα υπολογιστών δεν μπορούν να χρησιμοποιηθούν σε παράλληλα συστήματα, λόγω των χρονοβόρων υπολογισμών που εμπεριέχουν.

Από την άλλη μεριά, η απλότητα των διαδρομητών είναι απαραίτητη για δύο λόγους. Πρώτον, γιατί ως ένα βαθμό εξασφαλίζει την ταχύτητα και δεύτερον, γιατί μπορεί να κρατήσει το κόστος σε χαμηλά επίπεδα (τέτοιοι διαδρομητές θα χρησιμοποιηθούν σε όλους τους επεξεργαστές οι οποίοι σε αριθμό μπορεί να είναι και χιλιάδες).

Οι παραπάνω απαιτήσεις θέτουν και ορισμένους βασικούς κανόνες οι οποίοι κατευθύνουν τη σχεδίαση του δικτύου διασύνδεσης:

- Σε αντίθεση με ένα δίκτυο υπολογιστών, το δίκτυο διασύνδεσης έχει προκαθορισμένη μορφή η οποία αποφασίζεται από την αρχή. Δεν πρόκειται, δηλαδή, να υπάρξει αργότερα μεταβολή στην τοπολογία του.

- Επίσης, το δίκτυο είναι σημαντικό να έχει όσο το δυνατόν μεγαλύτερη συμμετρία, ώστε να εξασφαλιστεί το χαμηλό κόστος των διαδρομητών. Η λέξη «συμμετρία» σε γενικές γραμμές σημαίνει ότι από κάθε επεξεργαστή το δίκτυο «φαίνεται» ακριβώς το ίδιο. Με αυτόν τον τρόπο, αρκεί να σχεδιαστεί ένας μόνο διαδρομητής και ο ίδιος να χρησιμοποιηθεί σε κάθε επεξεργαστή.

- Τέλος, η τοπολογία του δικτύου θα πρέπει να έχει μορφή εύκολα περιγράψιμη, κάτι που οδηγεί στην απλή και δομημένη σχεδίαση των διαδρομητών καθώς επίσης και των αλγορίθμων επικοινωνίας.

3.2 Η τοπολογία του δικτύου

Θεωρώντας τον κάθε κόμβο ως *κορυφή* και κάθε σύνδεση ως *ακμή*, ένα δίκτυο διασύνδεσης μοντελοποιείται απλά ως ένας γράφος. Με αυτόν τον τρόπο καθορίζουμε την *τοπολογία* του δικτύου, η οποία καθορίζει την κατανομή των κόμβων στον χώρο και τις συνδέσεις μεταξύ τους. Η θεωρία γράφων μπορεί να χρησιμοποιηθεί στη συνέχεια προκειμένου να εξαχθούν ποσοτικά χαρακτηριστικά και ιδιότητες του δικτύου.

Προκειμένου να παρουσιάσουμε μερικά από τα συνήθη δίκτυα διασύνδεσης, θα κάνουμε πρώτα μία σύνοψη κάποιας βασικής ορολογίας της θεωρίας γράφων:

- Δύο κόμβοι (κορυφές) με άμεση σύνδεση (ακμή) μεταξύ τους ονομάζονται *γειτονικοί*.

- Το πλήθος των γειτόνων ενός κόμβου είναι ο *βαθμός* του. Αν όλοι οι κόμβοι έχουν τον ίδιο βαθμό, ο γράφος ονομάζεται *τακτικός*.

- Μία *διαδρομή* από έναν κόμβο a σε έναν κόμβο b χρησιμοποιεί ενδιάμεσους κόμβους και ακμές στον γράφο. Μετρώντας το *μήκος* της διαδρομής ως το πλήθος των ακμών που διασχίζει, η *απόσταση* μεταξύ των δύο κόμβων είναι το μήκος της συντομότερης διαδρομής που υπάρχει μεταξύ των.

- Η *διάμετρος* είναι η μέγιστη απόσταση που υπάρχει μέσα σε έναν γράφο.

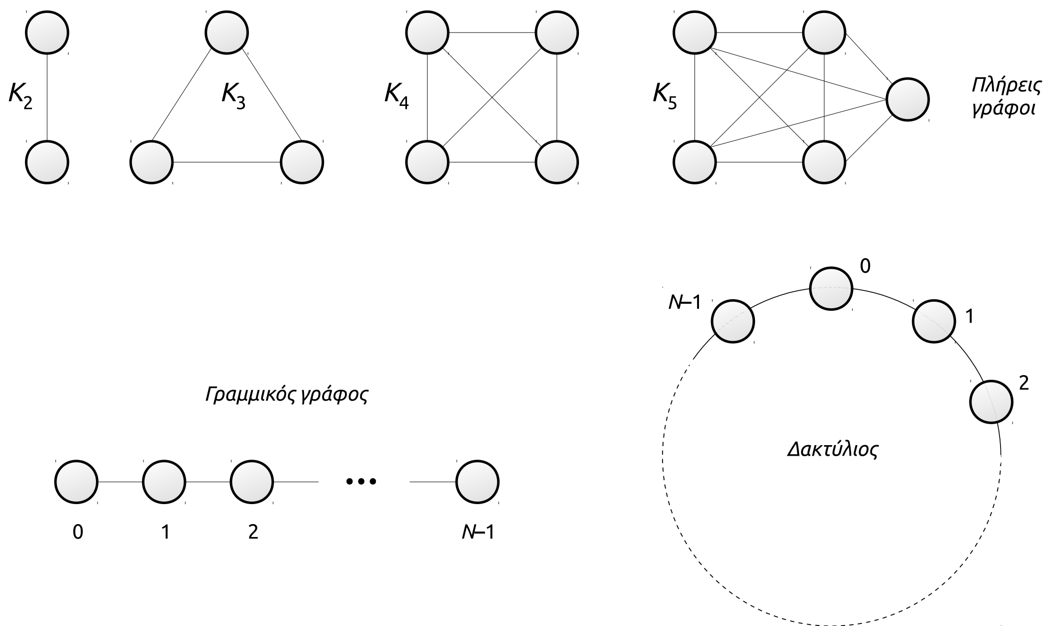
Τα παραπάνω γραφοθεωρητικά χαρακτηριστικά έχουν μεγάλη πρακτική σημασία. Πιο συγκεκριμένα, επειδή οι ακμές του γράφου αντιστοιχούν σε άμεσες συνδέσεις μεταξύ κόμβων και όχι σε ένα κοινόχρηστο μέσο, όπως ο δίαυλος, όσο μεγαλύτερο βαθμό έχει ένας κόμβος τόσο περισσότερα κανάλια εισόδου / εξόδου θα έχει ο διαδρομητής. Ο μεγαλύτερος βαθμός, επομένως, αυξάνει την πολυπλοκότητα του διαδρομητή, κάτι το οποίο έχει επιπτώσεις και στο κόστος του.

Ο αυξημένος βαθμός οδηγεί, επίσης, και στην ύπαρξη πολλαπλών διαδρομών μεταξύ δύο κόμβων. Ο διαδρομητής, ανάλογα με τον αλγόριθμο που χρησιμοποιεί, θα χρειαστεί πιθανώς επιπλέον λογική για την επιλογή των εναλλακτικών διαδρομών, αυξάνοντας περαιτέρω τις απαιτήσεις από το υλικό.

Από την άλλη μεριά, ο αυξημένος βαθμός συνδέεται και με τη βελτίωση της *αξιοπιστίας* του δικτύου. Λόγω της ύπαρξης περισσότερων εναλλακτικών μονοπατιών, αν κάποια ακμή καταστεί ανενεργή (π.χ. λόγω βλάβης), θα μπορούν να επιλεγούν κάποιες άλλες διαδρομές, οι οποίες αποφεύγουν τη συγκεκριμένη ακμή, εξασφαλίζοντας την αδιάκοπη λειτουργία του συστήματος.

Η διάμετρος, είναι ένα άλλο μέγεθος του γράφου με σημαντική πρακτική σημασία. Η διάμετρος καθορίζει τη μέγιστη απόσταση που υπάρχει μεταξύ δύο οποιονδήποτε κορυφών. Σε ένα δίκτυο διασύνδεσης, η διάμετρος συνδέεται με τη μέγιστη καθυστέρηση που μπορεί να υποστεί κάποιο μήνυμα από τη στιγμή που θα ξεκινήσει από έναν κόμβο μέχρι να φτάσει στον προορισμό του. Είναι, επομένως, επιθυμητό τα δίκτυα να διαθέτουν όσο το δυνατόν μικρότερη διάμετρο. Η αύξηση του βαθμού είναι πολλές φορές υπεύθυνη για τη μείωση της διαμέτρου.

Όπως είπαμε και στην εισαγωγή του κεφαλαίου, σε ένα δίκτυο διασύνδεσης ενδιαφερόμαστε και για την ταχύτητα επικοινωνίας, αλλά και για την απλότητα (άρα το χαμηλό κόστος) σχεδιασμού των διαδρομητών. Θα πρέπει να είναι, πλέον, κατανοητό ότι απαιτείται μια συμβιβαστική λύση μεταξύ βαθμού, διαμέτρου και αξιοπιστίας στο δίκτυο. Κανένα δίκτυο από όσα έχουν προταθεί στη βιβλιογραφία δεν αριστεύει σε όλα τα χαρακτηριστικά. Δεν είναι εύκολο να πετύχουμε και μικρό βαθμό (για απλότητα και μικρό κόστος) και μικρή διάμετρο (για ταχύτητα και αξιοπιστία).



Σχήμα 3.3 *Βασικές τοπολογίες.*

3.2.1 Βασικές τοπολογίες

Στο Σχ. 3.3, δίνονται μερικές από τις απλές αλλά κλασικές και χρήσιμες τοπολογίες δικτύων διασύνδεσης. Ο *πλήρης γράφος* K_N έχει όλες τις δυνατές συνδέσεις μεταξύ των N κόμβων του. Διαθέτει επομένως $N(N-1)/2$ συνδέσεις συνολικά, ενώ ο βαθμός του κάθε κόμβου είναι ίσος με $N-1$, καθώς συνδέεται με ακμή με όλους τους υπόλοιπους κόμβους. Για τον ίδιο λόγο, όλες οι αποστάσεις και επομένως, και η διάμετρός του K_N είναι ίσες με 1. Ο πλήρης γράφος αποτελεί ακραία τοπολογία, καθώς σε σχέση με οποιοδήποτε άλλο δίκτυο με N κόμβους, διαθέτει τον μέγιστο δυνατό βαθμό, το μέγιστο δυνατό πλήθος συνδέσεων και την ελάχιστη δυνατή διάμετρο. Αυτό τον κάνει έναν από τους ταχύτερους γράφους που όμως είναι πολύ δύσκολο να υλοποιηθεί για μεγάλο N, λόγω του μεγάλου βαθμού. Επιπλέον δεν *κλιμακώνεται* εύκολα, καθώς η προσθήκη έστω και ενός νέου κόμβου, απαιτεί την προσθήκη επιπλέον καναλιών σε όλους τους υπάρχοντες κόμβους.

Στον αντίποδα βρίσκονται τα δίκτυα του *γραμμικού γράφου* και *δακτυλίου*. Ειδικά ο γραμμικός γράφος διαθέτει τον ελάχιστο δυνατό αριθμό ακμών, με κάθε κόμβο να έχει βαθμό το πολύ ίσο με 2, και τη μέγιστη δυνατή διάμετρο ($N-1$ ακμές για τη διαδρομή από τον κόμβο 0 στον κόμβο $N-1$). Ο δακτύλιος είναι η συμμετρική και τακτική εκδοχή του γραμμικού γράφου όπου η επιπλέον ακμή που κλείνει τον κύκλο, μειώνει τη διάμετρο περίπου στο μισό ($\lfloor N/2 \rfloor$). Πρόκειται για τις οικονομικότερες και ταυτόχρονα πιο αργές τοπολογίες.

Τα παραπάνω δίκτυα, όπως είναι αναμενόμενο, δεν χρησιμοποιούνται συχνά αυτούσια σε μεγάλα συστήματα. Ο πλήρης γράφος είναι χρήσιμος για μικρά συστήματα, ενώ ο

γραμμικός γράφος έχει χρησιμοποιηθεί περισσότερο σε συστήματα ειδικού σκοπού, όπως για παράδειγμα συστολικές διατάξεις (systolic arrays). Ο δακτύλιος είναι ίσως το πιο συνηθισμένο από τα τρία δίκτυα. Ένα πολύ πρόσφατο παράδειγμα όπου γίνεται χρήση του δακτυλίου, είναι ο πολυπύρηνος επιταχυντής της Intel, ο Xeon Phi.

3.2.2 Σύνθετες τοπολογίες

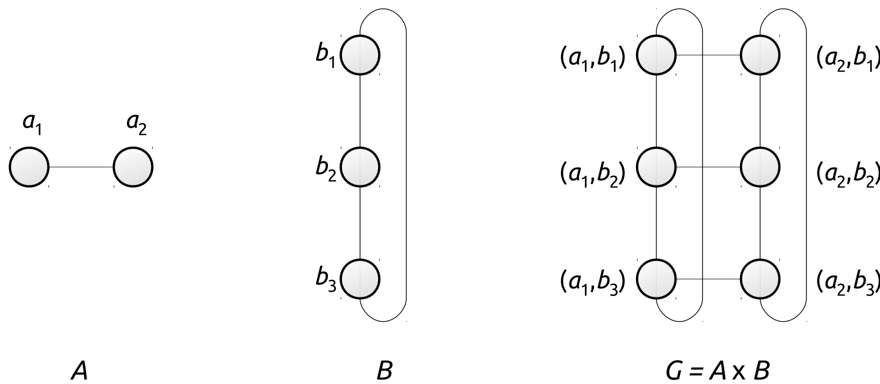
Τα βασικά δίκτυα της προηγούμενης ενότητας, αν και δεν χρησιμοποιούνται αυτούσια, αποτελούν τη βάση για τη δημιουργία συνθετότερων δικτύων, τα οποία παρουσιάζουν πιο ισορροπημένα χαρακτηριστικά. Υπάρχουν αρκετές μέθοδοι για τη *σύνθεση* δικτύων τα οποία επιθυμεί ο σχεδιαστής να έχουν προκαθορισμένες ιδιότητες, όπως για παράδειγμα:

- Η μεθοδολογία των γράφων ακμών, η οποία ξενικώντας από έναν γράφο παράγει έναν άλλο με βάση τις ακμές του πρώτου. Δίκτυα όπως τα Kautz και deBuijn είναι από τα πιο γνωστά αυτής της κατηγορίας.
- Η μέθοδος του Cayley που στηρίζεται σε αλγεβρικές δομές (ομάδες) και πράξεις μεταξύ των στοιχείων τους, προκειμένου να καθοριστούν οι ακμές. Οι γράφοι circulants, pancake, cube-connected cycles μπορούν να παραχθούν με τη μέθοδο Cayley.
- Η μεθοδολογία του καρτεσιανού γινομένου με την οποία θα ασχοληθούμε εδώ.

Το καρτεσιανό γινόμενο αποτελεί την πλέον δημοφιλή μέθοδο. Τα δίκτυα που προκύπτουν ονομάζονται *πολυδιάστατα* (multidimensional) και ορίζονται ως εξής. Ας υποθέσουμε ότι έχουμε δύο γράφους A και B με κορυφές $\{a_1, a_2, \ldots\}$ και $\{b_1, b_2, \ldots\}$ αντίστοιχα, και ας συμβολίσουμε με N_A (N_B) το πλήθος των κορυφών του A (B). Το καρτεσιανό τους γινόμενο $G = A \times B$ ορίζεται ως ένας γράφος με $N_G = N_A \times N_B$ κορυφές που αποτελούν το καρτεσιανό γινόμενο των συνόλων κορυφών του A και B. Με άλλα λόγια, οι κορυφές του G είναι οι:

$$\begin{gathered}\{(a_1, b_1), (a_1, b_2), \ldots, (a_1, b_{N_B})\\ (a_2, b_1), (a_2, b_2), \ldots, (a_2, b_{N_B}),\\ \vdots\\ (a_{N_A}, b_1), (a_{N_A}, b_2), \ldots, (a_{N_A}, b_{N_B})\}\end{gathered}$$

όπου η κάθε κορυφή ονοματίζεται με ένα ζεύγος που το πρώτο στοιχείο του («συντεταγμένη») προέρχεται από κορυφή του A και το δεύτερο από κορυφή του B. Ο γράφος G χαρακτηρίζεται ως διδιάστατος και οι δύο γράφοι A και B είναι οι δύο διαστάσεις του. Δύο κορυφές (x, y) και (z, w) είναι γειτονικές στον γράφο G, αν και μόνο αν οι κόμβοι έχουν ίδια τη μία συντεταγμένη και γειτονική την άλλη. Δηλαδή, είτε $y = w$ και τα x και z είναι γειτονικοί κόμβοι στον γράφο A, είτε $x = z$ και τα y και w είναι γειτονικοί κόμβοι στον B.



Σχήμα 3.4 *Καρτεσιανό γινόμενο δύο γράφων.*

Ένα παράδειγμα δίνεται στο Σχ. 3.4, όπου ο διδιάστατος γράφος G είναι το καρτεσιανό γινόμενο ενός γραμμικού γράφου K_2 και ενός δακτυλίου 3 κόμβων.

Πολλά χαρακτηριστικά του καρτεσιανού γινομένου προκύπτουν από τα χαρακτηριστικά των δύο γράφων-παραγόντων του. Αν συμβολίσουμε με $\beta_H(v)$ το βαθμό της κορυφής v στον γράφο H και D_H τη διάμετρο του γράφου, μπορεί να αποδειχτεί ότι:

$$\beta_G(a_i, b_i) = \beta_A(a_i) + \beta_B(b_i) \quad (3.1)$$

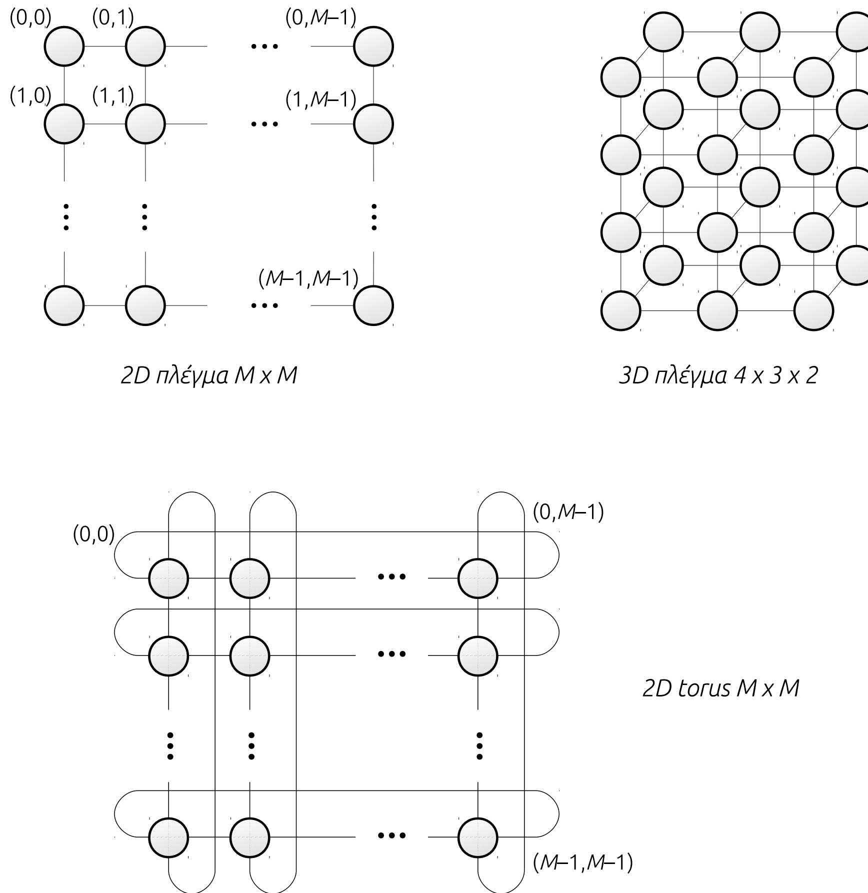
$$D_G = D_A + D_B \quad (3.2)$$

για οποιαδήποτε κορυφή (a_i, b_i) του G.

Με τη μέθοδο του καρτεσιανού γινομένου παράγονται μερικά από τα σπουδαιότερα δίκτυα διασύνδεσης. Μερικά από αυτά δίνονται στο Σχ. 3.5. Το διδιάστατο (2D) *πλέγμα* αποτελεί καρτεσιανό γινόμενο δύο γραμμικών γράφων. Αν αυτό πολλαπλασιαστεί με ένα ακόμα γραμμικό γράφο, προκύπτει ένα τριδιάστατο (3D) πλέγμα. Γενικεύοντας, μπορούμε να σχεδιάσουμε ένα d-διάστατο πλέγμα ως καρτεσιανό γινόμενο $d \geq 2$ γραμμικών γράφων, αν και κυρίως πλέγματα δύο και τριών διαστάσεων έχουν χρησιμοποιηθεί στην πράξη. Με βάση τις (3.2) και (3.2), μπορούμε να δούμε ότι σε ένα d-διάστατο πλέγμα ο βαθμός των κόμβων κυμαίνεται από d έως $2d$, ενώ το δίκτυο δεν είναι τακτικό. Σε ένα διδιάστατο πλέγμα $N = M \times M$ κόμβων, η διάμετρος είναι ίση με $2M - 2 \approx 2\sqrt{N}$.

Το γινόμενο δακτυλίων είναι γνωστό ως *torus* (βλ. Σχ. 3.5), και όπως και στα πλέγματα, μπορούμε να σχηματίσουμε tori οσωνδήποτε d διαστάσεων, πολλαπλασιάζοντας μεταξύ τους d δακτυλίους. Και εδώ, τα διδιάστατα και τριδιάστατα tori είναι τα πιο συνηθισμένα στην πράξη. Ως δίκτυο είναι συμμετρικό και τακτικό, βαθμού $2d$. Στο σχήμα, το διδιάστατο torus $N = M \times M$ έχει βαθμό κόμβων ίσο με 4 και διάμετρο $2\lfloor M/2 \rfloor \approx \sqrt{N}$. Αν και καλύτερο δίκτυο από πολλές πλευρές (όχι όμως όλες) σε σχέση με το πλέγμα, εντούτοις, το πλέγμα είναι αυτό που συναντά κανείς συχνότερα σε υλοποιημένα συστήματα.

Τέλος, ο *υπερκύβος* (hypercube) είναι ένα ιδιαίτερο δίκτυο το οποίο ορίζεται ως το καρτεσιανό γινόμενο γράφων K_2 με δύο κόμβους. Ο διδιάστατος κύβος Q_2 έχει το σχήμα



Σχήμα 3.5 *Σύνθετες τοπολογίες.*

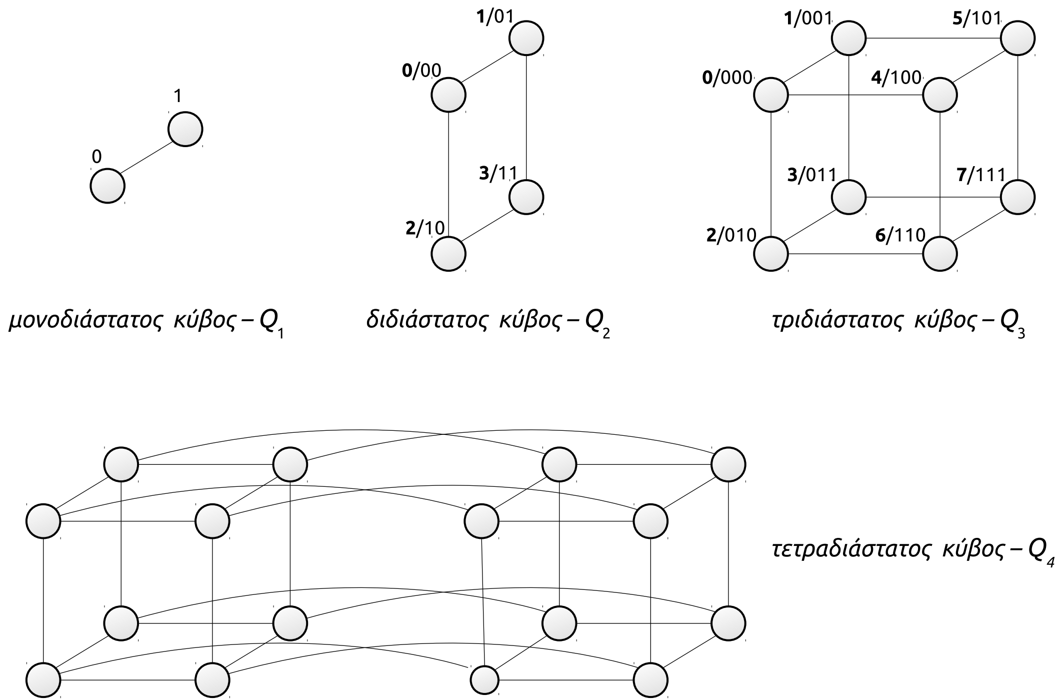
του τετραγώνου, ενώ ο τριδιάστατος Q_3 έχει το γεωμετρικό σχήμα του κύβου (Σχ. 3.6). Ο d-διάστατος υπερκύβος ή απλά d-κύβος ορίζεται αναδρομικά ως:

$$Q_d = Q_{d-1} \times K_2,$$

δηλαδή ο d-διάστατος υπερκύβος προκύπτει από το καρτεσιανό γινόμενο ενός $(d-1)$-διάστατου υπερκύβου και ενός K_2. Ακόμα γενικότερα ισχύει ότι:

$$Q_d = Q_{d_1} \times Q_{d_2},$$

αρκεί $d_1 + d_2 = d$. Ένας ισοδύναμος, αλλά πολλές φορές χρησιμότερος ορισμός είναι ο εξής: ο υπερκύβος Q_d αποτελείται από 2^d κόμβους κάθε ένας εκ των οποίων ονοματίζεται με έναν από τους 2^d d-ψήφιους δυαδικούς αριθμούς από το $(00\cdots0)_2$ έως το $(11\cdots1)_2$. Δύο κόμβοι είναι γειτονικοί αν και μόνο αν τα ονόματά τους διαφέρουν σε ακριβώς 1 bit. Στο Σχ. 3.6, φαίνεται τόσο η αναδρομική κατασκευή των κύβων, όσο και τα ονόματα των κόμβων. Αν $N = 2^d$, προκύπτει από τις (3.2) και (3.2) ότι τόσο ο βαθμός των κορυφών όσο και η διάμετρος του Q_d ισούνται με d, ή ισοδύναμα με $\log_2 N$.



Σχήμα 3.6 *Υπερκύβοι.*

Ο υπερκύβος είναι ίσως, το περισσότερο μελετημένο από τα παραπάνω δίκτυα. Έχει πλούσιες τοπολογικές ιδιότητες και έχει χρησιμοποιηθεί σε πολλά υλοποιημένα συστήματα. Έχει όμως και κάποια μειονεκτήματα, τα οποία αποδεικνύονται σημαντικά στην πράξη. Ένα, είναι ο αριθμός των κόμβων ο οποίος θα πρέπει να είναι πάντα δύναμη του 2. Ένα άλλο μειονέκτημα, είναι ο λογαριθμικός βαθμός των κορυφών. Ο βαθμός αυξάνει όσο αυξάνει ο αριθμός των κόμβων και γρήγορα γίνεται αρκετά μεγάλος, οδηγώντας σε αυξημένο κόστος υλοποίησης. Εκτός αυτού, είναι τέτοια η δομή του δικτύου που δεν επιτρέπει εύκολες αναβαθμίσεις. Αν θελήσουμε, σε ένα υπάρχον σύστημα, να προσθέσουμε νέους κόμβους για να δημιουργήσουμε μεγαλύτερο υπερκύβο, θα πρέπει να επηρεαστούν όλοι οι υπάρχοντες διαδρομητές, ώστε να προστεθούν νέα κανάλια προς τους νέους γείτονες. Κάτι τέτοιο δεν ισχύει π.χ. στο torus όπου για προσθήκη επιπλέον γραμμών / στηλών δεν χρειάζεται να τροποποιηθεί κανένας διαδρομητής, αφού έχουν πάντα ακριβώς τέσσερις συνδέσεις.

Στον Πίνακα 3.1 δίνονται συνοπτικά τα χαρακτηριστικά των δικτύων που μελετήσαμε παραπάνω. Εκτός από αυτά, έχουν προταθεί πολλά άλλα με διαφορετικά χαρακτηριστικά και ιδιότητες. Η έρευνα στον τομέα αυτό είναι αδιάκοπη αν και, όπως είπαμε πάλι, στην πράξη χρησιμοποιούνται σχεδόν αποκλειστικά μόνο κάποια από τα παραπάνω δίκτυα. Μέχρι περίπου τα τέλη της δεκαετίας του 1990, ο υπερκύβος ήταν το δίκτυο που μονοπωλούσε το χώρο των συστημάτων κατανεμημένης μνήμης. Από εκεί και μετά, τα πλέγματα και τα tori χαμηλών διαστάσεων (και κυρίως τα πλέγματα 2D) είναι αυτά που χρησιμοποιούνται ευρέως.

Πίνακας 3.1 *Σύνοψη χαρακτηριστικών από διάφορες τοπολογίες*

	Κόμβοι	**Βαθμός**	**Διάμετρος**	**Παραδείγματα συστημάτων**
Πλήρης γράφος	N	$N-1$	1	
Γραμμ. γράφος	N	$1,2$	$N-1$	
Δακτύλιος	N	2	$\lfloor N/2 \rfloor$	Sequent Symmetry, Intel Xeon Phi
Πλέγμα 2D	$N = M \times M$	$2,3,4$	$2M-2 \approx 2\sqrt{N}$	Intel Paragon Adapt. Epiphany
Πλέγμα 3D	$N = M \times M \times M$	$2,3,\ldots,6$	$3M-3 \approx 3\sqrt[3]{N}$	MIT J Machine, Sandia Red Storm
Torus 2D	$N = M \times M$	4	$2\lfloor M/2 \rfloor \approx \sqrt{N}$	Cray X1E
Torus 3D	$N = M \times M \times M$	6	$3\lfloor M/2 \rfloor \approx \sqrt[3]{N}$	Cray XT3/4/5/6 Cray XE6
Υπερκύβος	$N = 2^d$	$d = \log_2 N$	$d = \log_2 N$	Intel iPSC-1/2, nCUBE-1/2/3, SGI Origin

3.3 Αντιστοιχίσεις τοπολογιών

Πολλές φορές είναι χρήσιμο να αντιστοιχίσουμε κάποιο δίκτυο σε ένα άλλο. Ο λόγος είναι ότι κάποιος αλγόριθμος που έχει σχεδιαστεί για ένα δίκτυο A μπορεί να ζητηθεί να δουλέψει σε ένα δίκτυο B, διότι το δίκτυο B είναι αυτό που διαθέτουμε. Υπάρχουν πολλοί αλγόριθμοι (ειδικά για αριθμητικά και επιστημονικά προβλήματα) οι οποίοι είναι σχεδιασμένοι και βελτιστοποιημένοι για δίκτυα που έχουν τοπολογία δακτυλίου ή πλέγματος, για παράδειγμα.

Ένας από τους λόγους για τους οποίους ο υπερκύβος θεωρείται μία από τις ισχυρότερες διασυνδέσεις, είναι ότι εμπεριέχει πολλά άλλα δίκτυα. Συγκεκριμένα, στη δομή του μπορεί κανείς να βρει αυτούσιους γραμμικούς γράφους, δακτυλίους, καθώς και πλέγματα και tori. Επομένως, ένα πρόγραμμα που έχει σχεδιαστεί για μία από αυτές τις τοπολογίες, μπορεί να τρέξει χωρίς αλλαγές σε έναν υπερκύβο καταλλήλων διαστάσεων.

Ένας γράφος A που αντιστοιχίζεται σε κάποιον άλλο γράφο B, ονομάζεται *φιλοξενούμενος* (guest), ενώ ο B ονομάζεται *οικοδεσπότης* (host). Υπάρχουν διάφορα μεγέθη που καθορίζουν την ποιότητα της αντιστοίχισης, τα οποία περιλαμβάνουν:

- την *επέκταση* (expansion) η οποία ορίζεται από τον λόγο του αριθμού των κόμβων του οικοδεσπότη προς τον αριθμό των κόμβων του φιλοξενούμενου,
- το *φορτίο* (load) το οποίο είναι ο μέγιστος αριθμός κόμβων του φιλοξενούμενου

που απεικονίζεται σε έναν κόμβο του οικοδεσπότη,

- τη *συμφόρηση* (congestion) που είναι ο μέγιστος αριθμός ακμών του φιλοξενούμενου που αντιστοιχίζονται σε μία ακμή του οικοδεσπότη και
- τη *διαστολή* (dilation) η οποία είναι ο μέγιστος αριθμός συνεχόμενων ακμών του οικοδεσπότη στις οποίες απεικονίζεται μία ακμή του φιλοξενούμενου.

Υπό κανονικές συνθήκες (δηλαδή αν οι δύο γράφοι έχουν τον ίδιο αριθμό κόμβων), η επέκταση και το φορτίο της αντιστοίχισης είναι ίσα με 1. Αν όμως ο φιλοξενούμενος γράφος έχει περισσότερες κορυφές, τότε αναγκαστικά μία κορυφή του οικοδεσπότη θα «εξομοιώνει» παραπάνω από μία κορυφές του φιλοξενούμενου, και άρα το φορτίο θα είναι μεγαλύτερο του 1.

Εάν ο φιλοξενούμενος έχει τοπολογία η οποία μπορεί να βρεθεί αυτούσια ως υπογράφος του οικοδεσπότη, τότε η συμφόρηση και η διαστολή της απεικόνισης θα είναι ίσες με 1, αφού θα υπάρχει μία προς μία αντιστοίχιση ακμών. Σε αντίθετη περίπτωση, κάποια ακμή του φιλοξενούμενου μπορεί να χρειαστεί να «εξομοιωθεί» από ένα μονοπάτι στον οικοδεσπότη, αυξάνοντας έτσι τη διαστολή σε τιμή μεγαλύτερη της μονάδας.

Ένα πολύ απλό παράδειγμα είναι η αντιστοίχιση ενός γραμμικού γράφου με M κόμβους σε έναν δακτύλιο με N κόμβους. Πρέπει να είναι ξεκάθαρο ότι ο δακτύλιος περιέχει το απλό μονοπάτι ως υπογράφο. Επομένως, αν $N \geq M$, η απεικόνιση θα έχει φορτίο, συμφόρηση και διαστολή ίσα με 1.

Θα πρέπει να φαντάζεται κανείς ότι αν ο οικοδεσπότης εκτελεί κάποιο πρόγραμμα σχεδιασμένο για τον φιλοξενούμενο γράφο, τότε το αυξημένο φορτίο επιφέρει καθυστερήσεις στην εκτέλεση, αφού τη δεδομένη εργασία καλούνται να την εκτελέσουν λιγότεροι επεξεργαστές. Όμοια, αυξημένη διαστολή σημαίνει επίσης καθυστερήσεις, αφού οι τυχόν επικοινωνίες σε μία ακμή του φιλοξενούμενου θα αναγκάζονται πλέον να πραγματοποιούνται επάνω σε ολόκληρα μονοπάτια στον οικοδεσπότη. Με άλλα λόγια, το ζητούμενο είναι πάντα τα παραπάνω μεγέθη να είναι όσο το δυνατόν πιο κοντά στη μονάδα.

3.3.1 Αντιστοίχιση γραμμικών γράφων και δακτυλίων στον υπερκύβο

Ας υποθέσουμε ότι έχουμε ένα δακτύλιο (ή γραμμικό γράφο) R_N (L_N), όπου $N = 2^d$. Η αντιστοίχιση των κόμβων του δακτυλίου με τους κόμβους του Q_d επιτυγχάνεται μέσω των κωδίκων Gray. Ένας κώδικας Gray είναι μια ακολουθία δυαδικών αριθμών, η οποία έχει την ιδιότητα ότι ο προηγούμενος από τον επόμενο αριθμό διαφέρουν σε ένα ακριβώς bit. Επίσης, ο τελευταίος αριθμός με τον πρώτο αριθμό της σειράς διαφέρουν σε ένα bit. Ως παράδειγμα, παρακάτω δίνεται ένας κώδικας Gray όπου φαίνονται οι ιδιότητες που αναφέραμε:

$$G_2 = \{00, 01, 11, 10\}.$$

Ο κώδικας αυτός είναι «διάστασης» 2, δηλαδή κάθε αριθμός αποτελείται από 2 bit. Οι κώδικες Gray κατασκευάζονται αναδρομικά. Έστω ότι θέλουμε τον κώδικα Gray με d bits. Τότε:

$$G_d = \left\{0G_{d-1}, 1G_{d-1}^R\right\},$$

όπου G_{d-1} είναι ο $(d-1)$-διάστατος κώδικας Gray και G_{d-1}^R είναι ο ίδιος κώδικας στον οποίον, όμως, η ακολουθία των αριθμών δίνεται με αντίστροφη σειρά, π.χ.

$$G_2^R = \{10, 11, 01, 00\}.$$

Ο συμβολισμός aG_{d-1} σημαίνει ότι σε κάθε στοιχείο του G_{d-1} τοποθετούμε μπροστά το bit a. Έτσι, ο κώδικας Gray με τρία ψηφία είναι ο:

$$\begin{aligned} G_3 &= \left\{0G_2, 1G_2^R\right\} \\ &= \{0\{00, 01, 11, 10\},\ 1\{10, 11, 01, 00\}\} \\ &= \{000, 001, 011, 010, 110, 111, 101, 100\}. \end{aligned}$$

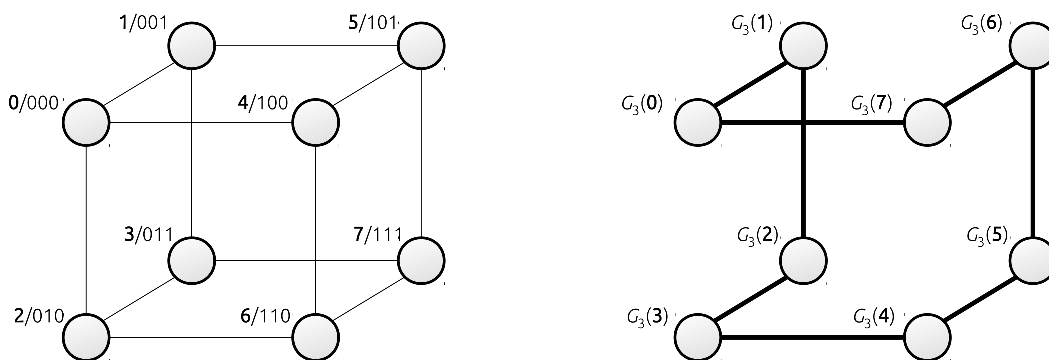
Έστω ότι $G_d(i)$ είναι το i-οστό στοιχείο (αριθμός) στον κώδικα G_d. Για παράδειγμα, $G_2(0) = 00$, $G_3(7) = 100$. Για να αντιστοιχίσουμε ένα γραμμικό γράφο L_N στον υπερκύβο, αντιστοιχίζουμε τον κόμβο i του L_N στον κόμβο $G_d(i)$ του υπερκύβου. Έτσι, για παράδειγμα, ένας γραμμικός γράφος L_8 θα έχει την εξής αντιστοίχιση στον Q_3:

Κόμβος στον L_8 ή R_8	0	1	2	3	4	5	6	7
Κόμβος στον Q_3	000	001	011	010	110	111	101	100

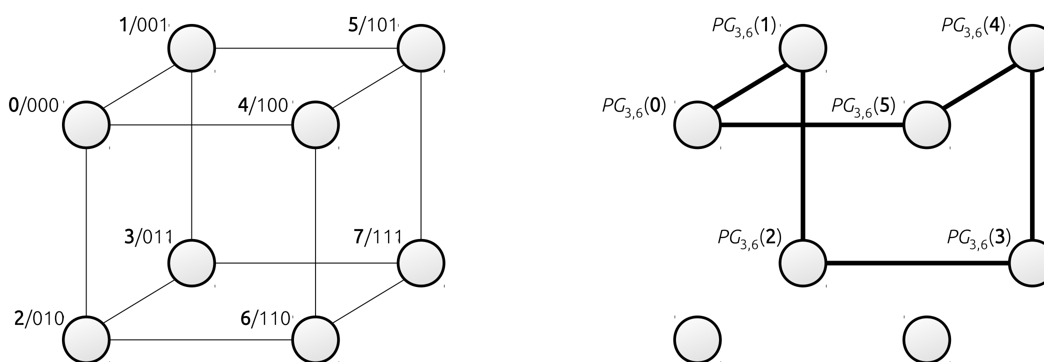
Η αντιστοίχιση αυτή είναι σωστή γιατί *διατηρούνται οι ακμές* του L_N. Με αυτό εννοούμε ότι, αν κάποιος κόμβος i στον γραμμικό γράφο έχει ένα γείτονα j, οι αντίστοιχοι κόμβοι στον υπερκύβο είναι επίσης γειτονικοί. Πράγματι, στον L_N, ο κόμβος i είναι γειτονικός με τον κόμβο $i+1$ και $i-1$ (για $i \neq 0, N-1$). Οι αντίστοιχοι κόμβοι στον Q_d είναι οι $G_d(i)$, $G_d(i+1)$ και $G_d(i-1)$. Όμως με βάση τις ιδιότητες του κώδικα Gray, γειτονικά στοιχεία στον κώδικα διαφέρουν κατά ένα bit. Δηλαδή, το $G_d(i)$ διαφέρει σε ένα bit από το $G_d(i+1)$ και το $G_d(i-1)$ διαφέρει σε ένα bit από το $G_d(i)$. Από τη στιγμή που οι διευθύνσεις δύο κόμβων διαφέρουν σε ένα bit, οι κόμβοι αυτοί πρέπει να είναι γείτονες στον υπερκύβο. Επομένως, ο κόμβος $G_d(i)$ γειτονεύει με τους $G_d(i-1)$ και $G_d(i+1)$ στον Q_d και άρα, κατά την αντιστοίχιση διατηρούνται οι ακμές του L_N.

Προσέξτε ότι το τελευταίο στοιχείο του κώδικα Gray διαφέρει σε ένα bit από το πρώτο στοιχείο του. Επομένως, οι κόμβοι $G_d(N-1)$ και $G_d(0)$ είναι γειτονικοί στον υπερκύβο. Το συμπέρασμα είναι ότι ακριβώς η ίδια αντιστοίχιση μπορεί να χρησιμοποιηθεί για να αντιστοιχίσουμε ένα δακτύλιο R_N στον υπερκύβο, όπως δόθηκε στον παραπάνω πίνακα. Στο Σχ. 3.7 δίνεται η αντιστοίχιση του R_8 στον Q_3.

Χρησιμοποιώντας τον «μερικό» κώδικα Gray με l στοιχεία, $PG_{d,l}$, μπορούμε να αντιστοιχίσουμε οποιονδήποτε γραμμικό γράφο ή δακτύλιο με $l < N$ κόμβους. Θυμηθείτε



Σχήμα 3.7 *Αντιστοίχιση του R_8 στον Q_3.*



Σχήμα 3.8 *Αντιστοίχιση του R_6 στον Q_3.*

ότι οι κύκλοι στον υπερκύβο έχουν πάντα άρτιο μήκος, επομένως, ένας δακτύλιος με l κόμβους μπορεί να αντιστοιχιστεί στον υπερκύβο μόνο αν το l είναι άρτιος. Αν $G_{d,m}$ είναι τα m πρώτα στοιχεία του G_d, ο μερικός κώδικας Gray ορίζεται ως:

$$PG_{d,l} = \left\{0G_{d-1,l/2}, 1G^R_{d-1,l/2}\right\}.$$

Για παράδειγμα, αν $l = 6$, τα 3 ($= l/2$) πρώτα στοιχεία του G_2 είναι:

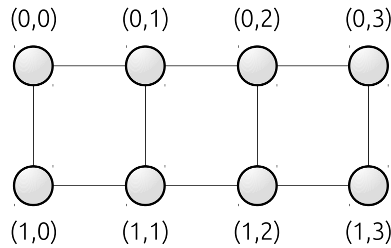
$$G_{2,3} = \{00, 01, 11\}$$

και επομένως, ο μερικός κώδικας Gray με 6 στοιχεία τριών ψηφίων είναι ο:

$$\begin{aligned} PG_{3,6} &= \left\{0G_{2,3}, 1G^R_{2,3}\right\} \\ &= \{000, 001, 011, 111, 101, 100\}. \end{aligned}$$

Ο κόμβος i του γραμμικού γράφου L_l ή του δακτυλίου R_l αντιστοιχίζεται στον κόμβο $PG_{d,l}(i)$ του Q_d. Στον επόμενο πίνακα και στο Σχ. 3.8 δίνεται η αντιστοίχιση για το παραπάνω παράδειγμα.

Κόμβος στον L_6 ή R_6	0	1	2	3	4	5
Κόμβος στον Q_3	000	001	011	111	101	100



Σχήμα 3.9 *Ένα πλέγμα* 2×4.

Το συμπέρασμα είναι ότι οποιοσδήποτε γραμμικός γράφος ή δακτύλιος με l κόμβους, όπου το l είναι άρτιος και $2 \leq l \leq 2^d$, μπορεί να αντιστοιχιστεί στον Q_d χρησιμοποιώντας τους κώδικες Gray. Η αντιστοίχιση έχει διαστολή, συμφόρηση και φορτίο ίσα με 1, αφού οι δακτύλιοι αυτοί αποτελούν, όπως είδαμε, υπογράφους του υπερκύβου.

3.3.2 Αντιστοίχιση πλεγμάτων και tori

Πλέγματα και tori μπορούν να αντιστοιχιστούν εύκολα στους κόμβους ενός υπερκύβου και η μέθοδος κάνει πάλι χρήση των κωδίκων Gray. Το γεγονός ότι οι διαστάσεις στα πλέγματα (tori) είναι γραμμικοί γράφοι (δακτύλιοι) οι οποίες αντιστοιχίζονται στον υπερκύβο, είναι ικανό να σιγουρέψει ότι τα πλέγματα και τα tori αντιστοιχίζονται επίσης στον υπερκύβο.

Ο υπερκύβος Q_d, όπως έχουμε πει, μπορεί να γραφεί και ως:

$$Q_d = Q_{d_1} \times Q_{d_2},$$

όπου $d = d_1 + d_2$.

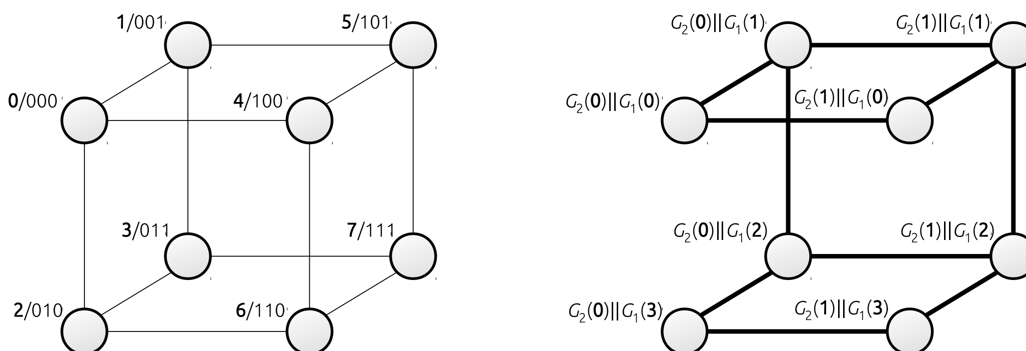
Ας υποθέσουμε ότι έχουμε ένα πλέγμα (ή torus) μεγέθους $2^{d_1} \times 2^{d_2}$. Όπως δείξαμε στην προηγούμενη ενότητα, ένας γραμμικός γράφος $L_{2^{d_i}}$ ή ένας δακτύλιος $R_{2^{d_i}}$ μπορεί να αντιστοιχιστεί σε έναν κύβο Q_{d_i}. Θα δούμε τώρα πώς το παραπάνω πλέγμα/torus μπορεί να αντιστοιχιστεί στον Q_d.

Η διαδικασία για την απεικόνιση πλεγμάτων/tori στον υπερκύβο απαιτεί πάλι τη χρήση των κωδίκων Gray. Θα χρησιμοποιήσουμε το σύμβολο '$\|$' για να δηλώσουμε τη συνένωση δύο σειρών από bits. Αν $x = (x_a x_{a-1} \cdots x_1)$ και $y = (y_b y_{b-1} \cdots y_1)$ είναι δύο δυαδικοί αριθμοί, τότε η συνένωσή τους είναι ένας δυαδικός αριθμός με $a + b$ bits:

$$x \parallel y = (x_a x_{a-1} \cdots x_1 y_b y_{b-1} \cdots y_1).$$

Έστω λοιπόν, ότι θέλουμε να αντιστοιχίσουμε το πλέγμα $2^{d_1} \times 2^{d_2}$ στον υπερκύβο Q_d, όπου $d = d_1 + d_2$. Η απεικόνιση γίνεται χρησιμοποιώντας δύο κώδικες Gray, με διαστάσεις d_1 και d_2· στη συνέχεια, ένας οποιοσδήποτε κόμβος (i, j) στο πλέγμα, αντιστοιχίζεται στον κόμβο:

$$G_{d_1}(i) \parallel G_{d_2}(j)$$



Σχήμα 3.10 *Αντιστοίχιση του πλέγματος* 2×4 *στον* Q_3.

του Q_d. Ως παράδειγμα, ας πάρουμε ένα πλέγμα 2×4, όπως φαίνεται στο Σχ. 3.9. Χρησιμοποιούμε τους κώδικες Gray με μία ($2 = 2^1$) και δύο ($4 = 2^2$) διαστάσεις,

$$\begin{aligned} G_1 &= \{0, 1\} \\ G_2 &= \{00, 01, 11, 10\}. \end{aligned}$$

Ο κόμβος $(0, 0)$ του πλέγματος αντιστοιχίζεται στον κόμβο $G_1(0) \parallel G_2(0) = 000$ του υπερκύβου, ο $(0, 1)$ αντιστοιχίζεται στον $G_1(0) \parallel G_2(1) = 001$ κλπ. Η πλήρης αντιστοίχιση δίνεται στον παρακάτω πίνακα και γραφικά στο Σχ. 3.10. Σε κάθε θέση του πίνακα δίνεται πρώτα ο κόμβος στο πλέγμα και δίπλα ο κόμβος στον υπερκύβο:

$(0,0) \to$ **000**	$(0,1) \to$ **001**	$(0,2) \to$ **011**	$(0,3) \to$ **010**
$(1,0) \to$ **100**	$(1,1) \to$ **101**	$(1,2) \to$ **111**	$(1,3) \to$ **110**

Προσέξτε ότι στις γραμμές του πίνακα, τα δύο τελευταία bits κάθε κόμβου του Q_d αποτελούν τον διδιάστατο κώδικα Gray. Επομένως, γειτονικοί κόμβοι σε μια γραμμή του πλέγματος είναι και γειτονικοί κόμβοι στον υπερκύβο. Σε κάθε στήλη, τα δύο τελευταία bits είναι ίδια, και το πρώτο bit σχηματίζει τον G_2. Επομένως, γειτονικοί κόμβοι σε μια στήλη του πλέγματος αντιστοιχίζονται σε γειτονικούς κόμβους στον υπερκύβο.

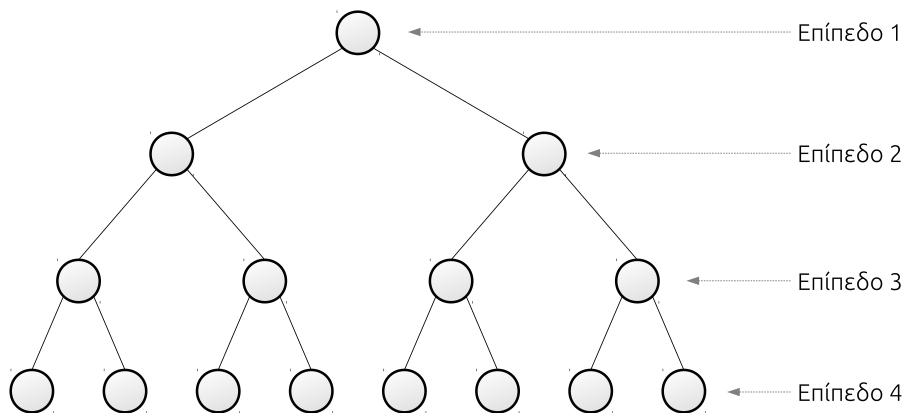
Γενικά, ένα πλέγμα ή torus $2^{d_1} \times 2^{d_2} \times \cdots \times 2^{d_k}$ απεικονίζεται στον Q_d, αντιστοιχίζοντας τον κόμβο (i_1, i_2, ..., i_k) του πλέγματος ή torus στον κόμβο

$$G_{d_1}(i_1) \parallel G_{d_2}(i_2) \parallel \cdots \parallel G_{d_k}(i_k)$$

του υπερκύβου.

Έχουμε δει ότι σε έναν d-κύβο μπορούμε να αντιστοιχίσουμε έναν οποιοδήποτε γραμμικό γράφο ή δακτύλιο με l κόμβους, αρκεί το l να είναι άρτιος και $2 \leq l \leq 2^d$ ή ισοδύναμα, $\lceil \log_2 l \rceil \leq d$. Χρησιμοποιώντας τους μερικούς κώδικες Gray και κάνοντας ακριβώς την ίδια διαδικασία όπως παραπάνω, δεν είναι ιδιαίτερα δύσκολο να δει κανείς ότι μπορούμε να αντιστοιχίσουμε οποιοδήποτε πλέγμα ή torus $l_1 \times l_2 \times \cdots \times l_k$ στον Q_d, αρκεί:

$$\lceil \log_2 l_1 \rceil + \lceil \log_2 l_2 \rceil + \cdots + \lceil \log_2 l_k \rceil \leq d.$$



Σχήμα 3.11 *Το πλήρες δυαδικό δέντρο με 4 επίπεδα.*

3.3.3 Αντιστοίχιση πλήρων δυαδικών δέντρων

Ένα δέντρο είναι δυαδικό, όταν κάθε κόμβος έχει το πολύ δύο παιδιά. Εάν όλοι οι κόμβοι, πλην των φύλλων, έχουν ακριβώς δύο παιδιά και επιπλέον, όλα τα φύλλα βρίσκονται στο ίδιο επίπεδο, τότε είναι ένα *πλήρες* δυαδικό δέντρο (ΠΔΔ). Στο Σχ. 3.11 δίνεται ένα ΠΔΔ με οκτώ φύλλα ή ισοδύναμα, τέσσερα επίπεδα. Ένα ΠΔΔ με d επίπεδα (μαζί με το επίπεδο της ρίζας) έχει ύψος $d-1$ και διαθέτει 2^{d-1} φύλλα. Γενικά, σε κάθε επίπεδο $i = 1, 2, \ldots$, υπάρχουν 2^{i-1} κόμβοι. Με έναν απλό υπολογισμό, βλέπουμε ότι οι κόμβοι στο ΠΔΔ είναι συνολικά:

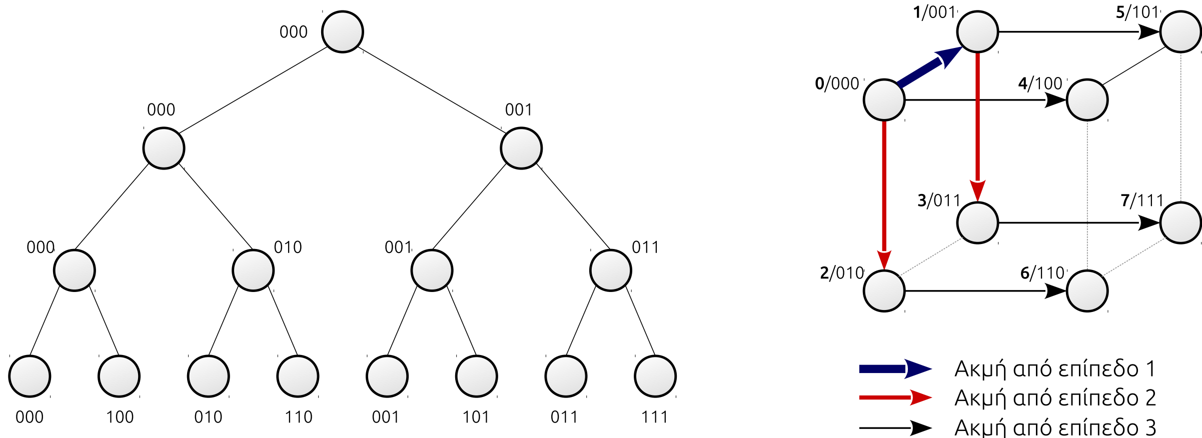
$$\sum_{i=1}^{d} 2^{i-1} = 2^d - 1.$$

Το ΠΔΔ είναι από τους πιο χρήσιμους γράφους, καθώς υπάρχει πληθώρα αλγορίθμων στους οποίους η ακολουθία των υπολογισμών τους έχει τη μορφή του δέντρου αυτού. Το πλήρες δυαδικό δέντρο, παρ' όλα αυτά, μπορεί να αποδειχτεί ότι δεν βρίσκεται αυτούσιο μέσα στον Q_d. Το ΠΔΔ με d επίπεδα μπορεί, όμως, να αντιστοιχιστεί στον υπερκύβο Q_d, με επέκταση $2^d/(2^d-1)$ και διαστολή 2 δηλαδή, θα υπάρχει κάποια ακμή του ΠΔΔ που αντιστοιχίζεται σε μονοπάτι μήκους 2 στον υπερκύβο. Η αντιστοίχιση αυτή δεν θα μας απασχολήσει εδώ.

Πολλές φορές ενδιαφερόμαστε να απεικονίσουμε μόνο τα φύλλα ενός ΠΔΔ με $d+1$ επίπεδα σε διαφορετικούς κόμβους στον Q_d. Σε αυτή την περίπτωση, ένας κόμβος στον υπερκύβο θα πρέπει να «ενσωματώνει» παραπάνω από έναν κόμβους του δέντρου (αφού το δέντρο έχει συνολικά $2^{d+1}-1$, ενώ ο Q_d έχει μόνο 2^d κόμβους). Μιλάμε επομένως, για μία απεικόνιση με επέκταση:

$$\frac{2^d}{2^{d+1}-1} \approx \frac{1}{2}.$$

Η απεικόνιση αυτή χρησιμοποιεί την παρακάτω διαδικασία:



Σχήμα 3.12 *Αντιστοίχιση του πλήρους δυαδικού δέντρου με 4 επίπεδα στον* Q_d.

- Αντιστοιχίζουμε τη ρίζα σε έναν οποιοδήποτε κόμβο του υπερκύβου.
- Για κάθε εσωτερικό κόμβο του δέντρου (δηλαδή εκτός από τα φύλλα) στο επίπεδο i, έστω x ο κόμβος στον οποίον αντιστοιχίζεται στον κύβο. Τότε, το αριστερό παιδί του αντιστοιχίζεται στον ίδιο κόμβο (x) και το δεξί παιδί του αντιστοιχίζεται στον κόμβο y του κύβου, ο οποίος προκύπτει από τον x, αν αντιστρέψουμε το i-οστό bit.

Στο Σχ. 3.12 δίνεται ένα παράδειγμα. Η ρίζα έχει αντιστοιχιστεί στον κόμβο 000 του κύβου. Από τη στιγμή που η ρίζα ανήκει στο *πρώτο* επίπεδο, το αριστερό παιδί της αντιστοιχίζεται επίσης στον 000 και το δεξί στον κόμβο 001 που προκύπτει από την αντιστροφή του *πρώτου* bit του 000. Προσέξτε ότι όλοι οι αριστερότεροι κόμβοι του δέντρου απεικονίζονται στον κόμβο 000 του υπερκύβου, πράγμα το οποίο δείχνει ότι το φορτίο της απεικόνισης είναι ίσο με $d+1$.

Κατά την απεικόνιση αυτή, από το ένα επίπεδο στο επόμενο αλλάζει μόνο το i-οστό bit. Επομένως, οι κόμβοι στον κύβο που περιλαμβάνουν τους κόμβους του επιπέδου i είτε περιέχουν κόμβους και του επόμενου επιπέδου είτε γειτονεύουν με αυτούς, και μάλιστα στην i-οστή διάσταση. Άρα, από όλους τους κόμβους ενός επιπέδου, μπορούμε να μεταφερθούμε στους κόμβους του επόμενου επιπέδου ταυτόχρονα, σε μόνο ένα βήμα στον υπερκύβο.

Ένα άλλο χαρακτηριστικό της απεικόνισης αυτής είναι ότι σε ένα επίπεδο, διαφορετικοί κόμβοι του δέντρου αντιστοιχίζονται σε διαφορετικούς κόμβους στον κύβο. Έτσι, αν ένας αλγόριθμος απαιτεί παράλληλη επεξεργασία ανάμεσα στους κόμβους ενός επιπέδου, αυτή γίνεται όντως παράλληλα στον υπερκύβο, αφού κάθε κόμβος του επιπέδου αυτού βρίσκεται σε διαφορετικό κόμβο του κύβου.

Συνοψίζοντας, είδαμε ότι ο υπερκύβος μπορεί να είναι οικοδεσπότης για πολλούς και χρήσιμους φιλοξενούμενους γράφους. Γραμμικοί γράφοι, δακτύλιοι, πλέγματα και tori

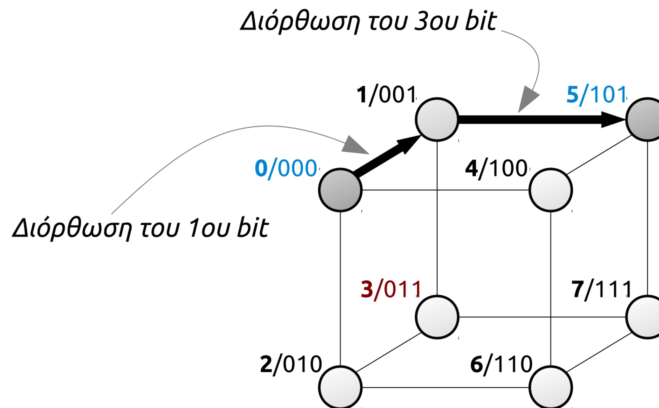
είναι μερικοί από αυτούς. Οι κώδικες Gray είναι το απαραίτητο «εργαλείο» για τις σωστές απεικονίσεις. Το πλήρες δυαδικό δέντρο δεν μπορεί όμως να αντιστοιχιστεί στον υπερκύβο με διαστολή 1. Παρ' όλα αυτά, μια πολύ εύχρηστη απεικόνιση μπορεί να υπάρξει, αν ενδιαφερθούμε να απεικονίσουμε τα φύλλα του δέντρου και όχι ολόκληρο το δέντρο σε διαφορετικούς κόμβους του κύβου. Η απεικόνιση αυτή εμφανίζεται πολύ συχνά στην πράξη και έχει χρήσιμες ιδιότητες.

3.4 Διαδρόμηση (Routing)

Η επικοινωνία μεταξύ δύο κόμβων είναι η βασικότερη μορφή επικοινωνίας σε ένα δίκτυο. Στην ενότητα αυτή θα δούμε σύντομα, διάφορες στρατηγικές που μπορούν να ακολουθηθούν προκειμένου να επιτευχθεί μεταφορά μηνύματος από έναν κόμβο σε έναν άλλο. Ουσιαστικά, το ζητούμενο είναι να βρεθεί μία διαδρομή στο δίκτυο μεταξύ των δύο κόμβων, πιθανώς ανάμεσα σε πολλές εναλλακτικές διαδρομές. Οι μέθοδοι που παρουσιάζουμε διαφέρουν στον τρόπο δημιουργίας της διαδρομής αυτής.

Υπάρχουν δύο τρόποι σχηματισμού της διαδρομής με βάση *ποιος κάνει την επιλογή*: είτε από τον πρώτο κόμβο που επιθυμεί να επικοινωνήσει είτε δυναμικά από όλους τους ενδιάμεσους κόμβους κατά τη διάρκεια σχηματισμού του μονοπατιού. Η διαδρόμηση ονομάζεται αντίστοιχα *πηγής* (source-based) ή *κατανεμημένη* (distributed). Στη διαδρόμηση πηγής ο κόμβος που στέλνει το μήνυμα έχει γνώση όλου του δικτύου και αποφασίζει εκ των προτέρων ποιους κόμβους και ποια κανάλια θα διασχίσει το μήνυμα μέχρι να φτάσει στον αποδέκτη του. Σε μία τέτοια περίπτωση όμως, το μήνυμα που θα μεταφερθεί αναγκαστικά θα περιέχει επιπλέον πληροφορίες για *ολόκληρη* τη διαδρομή που πρέπει να ακολουθηθεί, κάτι που πιθανώς είναι ανεπιθύμητο σε πολλές περιπτώσεις, ειδικά όταν τα μηνύματα είναι σχετικά μικρά. Η πιο συνηθισμένη τεχνική επομένως, είναι αυτή της κατανεμημένης διαδρόμησης, όπου ένας ενδιάμεσος κόμβος που λαμβάνει το μήνυμα αποφασίζει ο ίδιος ποιος θα είναι ο επόμενος κόμβος. Το μήνυμα θα πρέπει να αναφέρει απλά ποιος είναι ο προορισμός του.

Η διαδρόμηση μπορεί επίσης να είναι *σταθερή* (deterministic ή oblivious) ή *προσαρμοζόμενη* (non-deterministic ή adaptive), ανάλογα με το αν μεταξύ δύο κόμβων επιλέγεται πάντα ένα σταθερό μονοπάτι ή όχι. Στη δεύτερη περίπτωση, ο διαδρομητής σε ενδιάμεσους κόμβους μπορεί να επιλέξει μεταξύ πολλών εναλλακτικών γειτόνων για να προωθήσει ένα μήνυμα. Οι προσαρμοζόμενες στρατηγικές έχουν τη δυνατότητα να ρυθμίζουν τις διαδρομές ανάλογα με δυναμικές καταστάσεις του δικτύου (π.χ. αποφυγή μονοπατιών που έχουν αυξημένη κίνηση ή που εμπεριέχουν κόμβους ή ακμές που δεν λειτουργούν). Η πιο απλή, σταθερή διαδρόμηση που είναι και η πιο συνηθισμένη, επιλέγει πάντα την ίδια διαδρομή μεταξύ δύο κόμβων. Η πορεία δηλαδή, καθορίζεται μόνο από τη διεύθυνση της πηγής και



Σχήμα 3.13 *Διαδρόμηση στον 3-κύβο από τον κόμβο 0 στον κόμβο 5.*

του παραλήπτη.

Τέλος, η διαδρόμηση μπορεί να είναι *ελάχιστη* (minimal) ή όχι (non-minimal), ανάλογα με τα μήκη των διαδρομών που ακολουθούνται. Όταν η διαδρόμηση είναι ελάχιστη, τα επιλεγόμενα μονοπάτια επικοινωνίας έχουν πάντα το μικρότερο δυνατό μήκος. Στην αντίθετη περίπτωση, τα μονοπάτια μπορεί να έχουν μεγαλύτερα μήκη (η επιλογή τους συνήθως συνδυάζεται με μία προσαρμοζόμενη πολιτική διαδρόμησης). Στη μη-ελάχιστη διαδρόμηση είναι βασικό να αποφευχθεί το φαινόμενο όπου ένα μήνυμα γυρίζει διαρκώς στο δίκτυο χωρίς να φτάνει ποτέ στον προορισμό του. Για λόγους ταχύτητας και οικονομίας στο υλικό, σχεδόν αποκλειστικά χρησιμοποιούνται αλγόριθμοι διαδρόμησης οι οποίοι είναι κατανεμημένοι, σταθεροί και ελάχιστοι.

3.4.1 Παράδειγμα: διαδρόμηση σε d-διάστατο υπερκύβο

Μπορεί να αποδειχτεί ότι στον d-διάστατο υπερκύβο Q_d, δύο κόμβοι $a = (a_{d-1}a_{d-2}\cdots a_0)_2$ και $b = (b_{d-1}b_{d-2}\cdots b_0)_2$ έχουν απόσταση ίση με το πλήθος των bits στα οποία διαφέρουν. Με βάση αυτό, προκύπτει ο εξής αλγόριθμος διαδρόμησης: αν το μήνυμα βρίσκεται στον κόμβο a και προορίζεται για τον κόμβο b, ο a βρίσκει τα bits στα οποία διαφέρει από τον b και προωθεί το μήνυμα στον γείτονά του σε μία από τις διαστάσεις που υπάρχει διαφορά στα bits. Με άλλα λόγια, επιλέγει έναν από τους γείτονές του (με τους οποίους όπως γνωρίζουμε θα διαφέρει σε ακριβώς ένα bit), ο οποίος να διαφέρει από τον a σε ένα από τα bits που ο a διαφέρει και από τον b. Με αυτό τον τρόπο, «διορθώνεται», όπως λέγεται, ένα bit και η διαδρομή πλησιάζει ένα βήμα πιο κοντά στον προορισμό της.

Το παράδειγμα στο Σχ. 3.13 εξηγεί τη διαδρομή από τον κόμβο 0 ($= 000_2$) στον κόμβο 5 ($= 101_2$). Αρχικά, βλέπουμε ότι οι δύο κόμβοι διαφέρουν σε 2 bits συνολικά, και συγκεκριμένα στα δύο λιγότερο σημαντικά. Ο κόμβος 0 πρέπει να επιλέξει έναν από τους γείτονές του, ώστε να «διορθώσει» ένα από αυτά τα δύο bits. Επομένως, πρέπει να προωθήσει το μήνυμα είτε στον κόμβο 001_2 είτε στον 100_2 (και όχι στον 010_2). Ας υποθέσουμε

ότι επιλέγει τον πρώτο και άρα στέλνει το μήνυμα στον γείτονα $1 = 001_2$, χρησιμοποιώντας δηλαδή, την ακμή στην πρώτη διάσταση. Ο παραλήπτης κάνει ακριβώς τα ίδια: βρίσκει τα bits στα οποία διαφέρει αυτός από τον προορισμό του μηνύματος (μόνο στο αριστερό bit στη συγκεκριμένη περίπτωση). Επομένως, «διορθώνει» το bit αυτό, προχωρώντας στην τρίτη διάσταση και στέλνοντάς το στον κόμβο 101_2, ο οποίος είναι και ο τελικός προορισμός.

Η διαδρόμηση αυτή διορθώνει τα διαφορετικά bits ένα προς ένα καθώς σχηματίζεται το μονοπάτι, μέχρι να διορθωθούν όλα και να έρθουν στις τιμές που έχουν τα bits του παραλήπτη (δηλαδή μέχρι να φτάσουμε στον προορισμό μας). Από τη στιγμή που πρέπει να διορθωθούν συνολικά τόσα bits όσα διαφέρει η πηγή από τον προορισμό, και σε κάθε βήμα διορθώνεται ένα ακριβώς bit, ο αριθμός των βημάτων μέχρι την άφιξη στον προορισμό είναι ίσος με τον αριθμό αυτό των διαφορετικών bits, δηλαδή είναι ίσος με την απόσταση των δύο εμπλεκόμενων κόμβων στον υπερκύβο. Η παραπάνω διαδρόμηση είναι επομένως ελάχιστη. Ταυτόχρονα, είναι κατανεμημένη μιας και ο κάθε ενδιάμεσος κόμβος καθορίζει μόνος του ποιος θα είναι ο επόμενος κόμβος στη διαδρομή. Γίνεται επίσης και σταθερή αν αποφασίσουμε ότι η διόρθωση των bits θα γίνεται πάντα με μία προκαθορισμένη σειρά.

Στη διαδρόμηση *e-cube*, η οποία είναι ίσως η δημοφιλέστερη μέθοδος διαδρόμησης σε υπερκύβους, κάθε κόμβος διαλέγει να διορθώσει το πιο σημαντικό από τα bit στα οποία διαφέρει από τον προορισμό, για να προχωρήσει η διαδρομή. Στο παραπάνω παράδειγμα, αν ακολουθούσαμε αυτή τη στρατηγική, ο κόμβος 000_2 θα έπρεπε να επιλέξει τον κόμβο 100_2 και όχι τον 001_2 ως τον επόμενο κόμβο του μονοπατιού.

3.5 Μεταγωγή (Switching)

Εκτός από την τοπολογία και τη μέθοδο διαδρόμησης, πολύ σημαντικό ρόλο στη συμπεριφορά και τις επιδόσεις ενός δικτύου διασύνδεσης παίζει το είδος μεταγωγής που χρησιμοποιεί. Η μεταγωγή είναι ο μηχανισμός με τον οποίο μεταφέρονται τα μηνύματα από μια είσοδο ενός διαδρομητή σε μία έξοδό του. Τέσσερις είναι οι βασικές τεχνικές μεταγωγής που θα δούμε: μεταγωγή μηνύματος ή πακέτου, μεταγωγή virtual cut-through, μεταγωγή κυκλώματος και μεταγωγή wormhole.

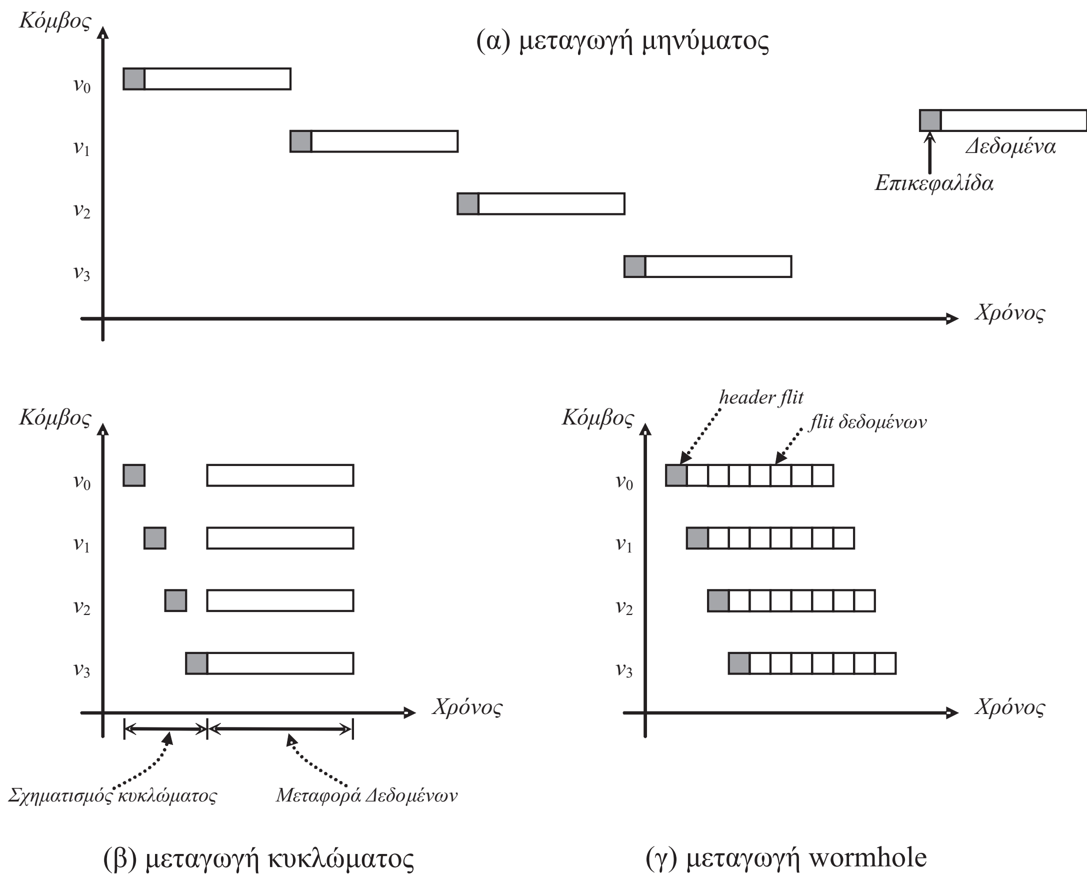
Στη *μεταγωγή μηνύματος* (message ή packet switching ή γενικότερα store-and-forward switching—SAF για συντομία), όταν ένα ω φτάνει σε έναν ενδιάμεσο κόμβο αποθηκεύεται τοπικά σε κάποιον buffer. Κατόπιν, το μήνυμα προωθείται σε έναν γειτονικό κόμβο, όταν το κανάλι διασύνδεσης είναι ελεύθερο και ο επόμενος κόμβος έχει χώρο για να αποθηκεύσει το μήνυμα. Η διαδικασία επαναλαμβάνεται μέχρι τον τελικό προορισμό. Επειδή τα μηνύματα μπορεί να είναι άλλοτε μικρά και άλλοτε μεγάλα, κάτι το οποίο δημιουργεί δυσκολία στον καθορισμό του απαιτούμενου αποθηκευτικού χώρου στους buffers, τα μηνύματα σχεδόν πάντα τμηματοποιούνται από την πηγή σε πακέτα σταθερού μήκους

τα οποία προωθούνται ανεξάρτητα στο δίκτυο και συναρμολογούνται πάλι σε ενιαίο μήνυμα στον προορισμό. Επομένως, οι ενδιάμεσοι διαδρομητές έχουν να χειριστούν πακέτα ενός συγκεκριμένου μεγέθους πάντα. SAF χρησιμοποιήθηκε κυρίως στους πρώτους παράλληλους υπολογιστές κατανεμημένης μνήμης, όπως ήταν οι Cosmic Cube, Intel iPSC-1, nCUBE-1, Ametek 14 και FPS T-series.

Προκειμένου να μειωθεί ο χρόνος που απαιτείται για την αποθήκευση ολόκληρου του μηνύματος ή του πακέτου στους κόμβους (ειδικά όταν είναι μεγάλο), προτάθηκε η *μεταγωγή virtual cut-through* (VCT), ή απλά *cut-through*. Καθώς το πακέτο φτάνει σε έναν ενδιάμεσο κόμβο, εξετάζονται άμεσα οι πληροφορίες στην «επικεφαλίδα» του η οποία είναι το τμήμα εκείνο που περιγράφει τον τελικό προορισμό του. Αν ο παραλήπτης είναι κάποιος άλλος κόμβος, τότε, πριν ολοκληρωθεί καν η λήψη ολόκληρου του πακέτου, αρχίζει αμέσως η προώθησή του στον επόμενο κόμβο. Με αυτόν τον τρόπο, αν όλα πάνε καλά, το μήνυμα δεν συναντά σημαντική καθυστέρηση και φτάνει ταχύτατα στον προορισμό του. Στην περίπτωση όμως που το κανάλι διασύνδεσης με τον επόμενο κόμβο είναι κατειλημμένο, το μήνυμα θα πρέπει να αποθηκευτεί και να προωθηθεί αργότερα. Επομένως, στη μεταγωγή VCT (όπως και στην SAF) δεν μπορούμε να αποφύγουμε τη χρήση μνήμης (buffers) για αποθήκευση μηνυμάτων. Η μεταγωγή VCT χρησιμοποιήθηκε στο ερευνητικό σύστημα Harts του Πανεπιστημίου του Michigan και χρησιμοποιείται και σε πολλά σύγχρονα συστήματα.

Εντελώς διαφορετική είναι η φιλοσοφία στη *μεταγωγή κυκλώματος* (circuit switching, CS). Εδώ, ένα φυσικό κύκλωμα κατασκευάζεται μεταξύ πηγής και προορισμού και μόλις ολοκληρωθεί, το μήνυμα προωθείται στον προορισμό απευθείας. Για τον σχηματισμό του κυκλώματος (circuit establishment phase) ένας ενδιάμεσος κόμβος επικοινωνεί με τον επόμενο κόμβο στη διαδρομή (στέλνει διερευνητικό σήμα, probe, όπως λέγεται), ζητώντας αποκλειστική σύνδεση. Η σύνδεση αυτή μόλις γίνει, αφιερώνεται στο ζευγάρι των κόμβων που πρέπει να επικοινωνήσουν και δεν μπορεί να χρησιμοποιηθεί για μεταφορά κανενός άλλου μηνύματος. Μόλις γίνει και η τελευταία σύνδεση προς τον προορισμό, δηλαδή ολοκληρωθεί το φυσικό κύκλωμα, αρχίζει η μεταφορά του μηνύματος (message transmission phase) το οποίο μεταφέρεται επάνω στο κύκλωμα αυτό κατευθείαν στον προορισμό. Οι ενδιάμεσοι κόμβοι δεν χρειάζεται, επομένως, να διαθέτουν μνήμη για την αποθήκευση του μηνύματος. Τέλος, αφού ολοκληρωθεί η μεταφορά του μηνύματος, το κύκλωμα ελευθερώνεται για να χρησιμοποιηθούν οι γραμμές από το υπόλοιπο δίκτυο (circuit termination phase). Οι iPSC-2 και iPSC/860 της Intel και ο Meiko CS-2 είναι παραδείγματα συστημάτων που χρησιμοποιούσαν μεταγωγή κυκλώματος.

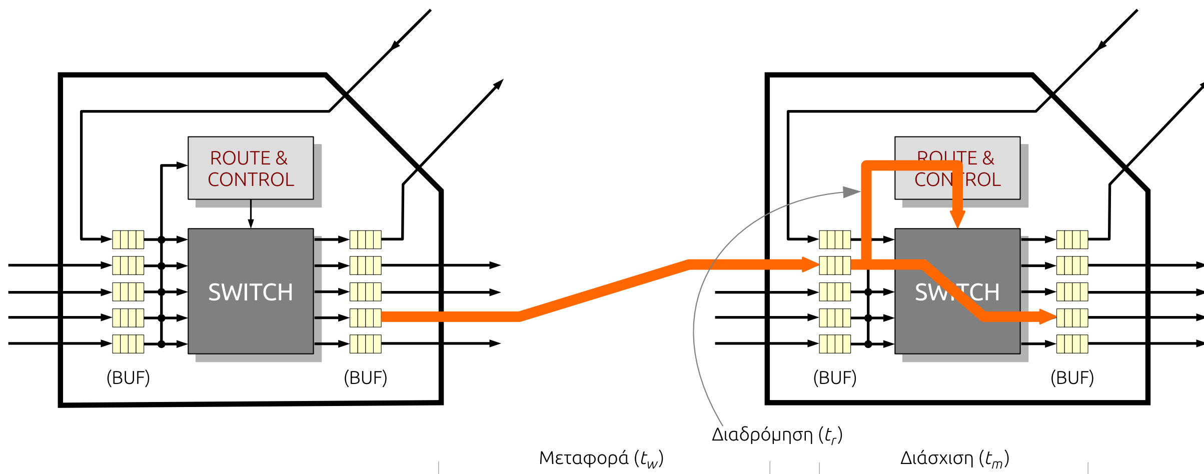
Η μεταγωγή wormhole (WS) μπορεί να θεωρηθεί ως είδος μεταγωγής VCT και βρίσκεται κάπου ανάμεσα στο SAF και το CS. Ένα μήνυμα διαιρείται σε πολύ μικρά πακέτα που ονομάζονται *flits* (flow control digits), από τα οποία το πρώτο (header flit) κατευθύνει όλο το μήνυμα. Καθώς το header flit προχωράει στο δίκτυο, εξετάζεται από τους ενδιάμεσους κόμβους και προωθείται αμέσως, όπως στο VCT. Τα υπόλοιπα flits ακολουθούν από πίσω, πάνω στη διαδρομή που χαράζει το πρώτο flit, και όλη η ακολουθία παρομοιάζεται



Σχήμα 3.14 *Μεταγωγή* SAF, CS *και* WS.

με κινούμενο σκουλήκι. Με αυτόν τον τρόπο, όπως στο CS, όταν το πρώτο flit φτάνει στον προορισμό, ακολουθούν τα flits του μηνύματος αμέσως το ένα πίσω από το άλλο. Αν όμως για κάποιον λόγο δεν μπορεί να προχωρήσει το header flit από έναν κόμβο στον επόμενο, σε αντίθεση με το VCT, το μήνυμα δεν αποσύρεται σε κάποια μνήμη. Αντίθετα, τα flit «μπλοκάρονται» στους κόμβους που βρίσκονται, καταλαμβάνοντας όλη τη μέχρι εκείνη τη στιγμή συμπληρωμένη διαδρομή (όπως στο CS). Επομένως, δεν απαιτούνται μεγάλες ποσότητες μνήμης για την αποθήκευση των μηνυμάτων, παρά μόνο ένας μικρός buffer για την αποθήκευση ενός flit σε κάθε κανάλι. Η μεταγωγή αυτή είναι από τις πιο δημοφιλείς και έχει χρησιμοποιηθεί σε συστήματα όπως τα NCUBE-2, Intel Touchtone Delta, Intel Paragon, MIT J Machine και άλλα.

Στο Σχ. 3.14, δίνεται σε απλουστευμένη μορφή η χρονική συμπεριφορά των διαφόρων τεχνικών μεταγωγής καθώς ένα μήνυμα περνά από τους κόμβους $v_0 - v_3$, θεωρώντας ότι το μήνυμα δεν συναντά κανένα εμπόδιο στο δρόμο του. Όπως φαίνεται και από το σχήμα, υπάρχει μία χρονική υπεροχή της μεταγωγής κυκλώματος και VCT/wormhole σε σχέση με τη μεταγωγή μηνύματος, καθώς αποφεύγεται η χρονοβόρα αποθήκευση ολόκλη-



Σχήμα 3.15 *Χρονική ανάλυση μεταφοράς 1 flit μεταξύ δύο κόμβων.*

ρου του μηνύματος στους ενδιάμεσους κόμβους. Στην αμέσως επόμενη ενότητα αναλύουμε με μεγαλύτερη λεπτομέρεια τον χρονισμό των τεχνικών.

3.5.1 Χρονική ανάλυση μεταγωγής

Ας υποθέσουμε ότι έχουν ένα κανάλι το οποίο λειτουργεί σε συχνότητα B Hz. Αυτό σημαίνει ότι, αν το πλάτος του καναλιού (δηλαδή το πόσα bits μπορεί να μεταφέρει ταυτόχρονα, γνωστό και ως phit) είναι w bits, τότε μεταφέρει Bw bps (bits-per-second), το οποίο ονομάζεται *εύρος ζώνης* (bandwidth) του καναλιού. Ισοδύναμα, ένα phit των w bits χρειάζεται χρόνο $t_w = 1/B$ για να μεταφερθεί.

Στο Σχ. 3.15, με έντονη γραμμή φαίνεται η πορεία που ακολουθεί ένα μήνυμα όταν διαπερνά έναν ενδιάμεσο κόμβο καθώς ταξιδεύει προς τον προορισμό του. Αρχικά, μεταφέρεται επάνω στο φυσικό κανάλι από τον προηγούμενο κόμβο προς ένα κανάλι εισόδου στον τρέχοντα κόμβο. Με βάση τα παραπάνω, w bits χρειάζονται χρόνο t_w για τη μεταφορά αυτή (χρόνος «μεταφοράς»). Εσωτερικά στον διαδρομητή, η μονάδα ελέγχου και διαδρόμησης θα πρέπει να πάρει απόφαση για το ποιο κανάλι εξόδου πρέπει να χρησιμοποιηθεί, με βάση τον αλγόριθμο διαδρόμησης που εφαρμόζει. Αυτό θα απαιτήσει χρόνο «διαδρόμησης» t_r. Τέλος, με t_m θα συμβολίσουμε το χρόνο που χρειάζεται μέχρι την έξοδο του μηνύματος από το διαδρομητή (χρόνος «διάσχισης»). Ο χρόνος αυτός κυριαρχείται κυρίως από τον χρόνο αποθήκευσης στους buffers (εισόδου ή/και εξόδου) του διαδρομητή, αλλά συμπεριλαμβάνουμε σε αυτόν και όποιες άλλες μικροκαθυστερήσεις, όπως π.χ. τον χρόνο διάσχισης του διασταυρωτικού διακόπτη.

Με βάση τα παραπάνω, μπορούμε να αναλύσουμε τη συμπεριφορά των τεχνικών μεταγωγής που είδαμε νωρίτερα. Θα υποθέσουμε ότι έχουμε να μεταφέρουμε ένα μήνυμα σε απόσταση D, δηλαδή θα πρέπει να περάσει από D ενδιάμεσους διαδρομητές πριν κα-

ταλήξει στον προορισμό του. Το μήνυμα δεν συναντά κανένα εμπόδιο στο δρόμο του και επομένως, δεν μελετάμε καταστάσεις όπου υπάρχουν αναμονές σε ενδιάμεσους κόμβους. Ως απλούστευση, η οποία δεν αλλάζει σε κάτι τη συνολική εικόνα, θα υποθέσουμε ότι 1 flit είναι ίσο με 1 phit, δηλαδή ότι 1 flit αποτελείται από w bits και θέλει χρονο $t_w = 1/B$ για να μεταφερθεί πάνω σε ένα φυσικό κανάλι. Το μήνυμα που θέλουμε να μεταφέρουμε, έχει μέγεθος M flits καθώς και 1 επιπλέον flit που θα αποτελέσει την επικεφαλίδα του, και η οποία χρησιμοποιείται από τους ενδιάμεσους κόμβους για τη διαδρόμηση. Επομένως, συνολικά το μήνυμα έχει $M + 1$ flits ή $L = (M + 1)w$ bits.

Μεταγωγή κυκλώματος Στη μεταγωγή κυκλώματος, το flit επικεφαλίδας παίζει τον ρόλο του probe στην πρώτη φάση σχηματισμού του κυκλώματος, προχωρώντας από κόμβο σε κόμβο, δεσμεύοντας κανάλια. Επομένως, θα χρειαστεί χρόνο t_w για να διασχίζει το κανάλι, t_r για τη διαδρόμηση μέσα σε έναν κόμβο, και επιπλέον χρόνο t_m για την τελικη προώθησή του στην έξοδο του διαδρομητή. Αυτό θα συμβεί ακριβώς D φορές, και επομένως, η φάση σχηματισμού του κυκλώματος απαιτεί χρόνο $D(t_w + t_r + t_m)$.

Με το σχηματισμό του φυσικού κυκλώματος, στέλνεται flit ειδοποίησης πίσω στον κόμβο-πηγή προκειμένου να ξεκινήσει την αποστολή του μηνύματος. Η ειδοποίηση αυτή, χρησιμοποιεί τα ήδη δεσμευμένα κανάλια προς τα πίσω, τα οποία έχουν σχηματίσει πλέον ένα φυσικό κύκλωμα. Επομένως, δεν θα υποστεί τις καθυστερήσεις για διαδρόμηση (μιας και έχουν ήδη καθοριστεί τα κανάλια εισόδου και εξόδου) και της ενδιάμεσης διάσχισης (t_m), καθώς τα κανάλια είναι ήδη συνδεδεμένα μεταξύ τους μέσω του διακόπτη crossbar και αποτελούν όλα μαζί ένα μεγάλο φυσικό κύκλωμα. Επομένως, θα χρειαστεί χρόνο t_w συνολικά.

Η τελευταία φάση είναι η μεταφορά του μηνύματος επάνω στο φυσικό κύκλωμα που έχει σχηματιστεί, πάλι χωρίς ενδιάμεσες καθυστερήσεις διαδρόμησης και διάσχισης, με απαιτούμενο χρόνο Mt_w για όλο το μήνυμα. Επομένως, ο συνολικός χρόνος για τη μεταφορά του μηνύματος με μεταγωγή κυκλώματος είναι ίσος με:

$$T_{CS} = D(t_w + t_r + t_m) + (M + 1)t_w.$$

Μεταγωγή SaF Στη μεταγωγή SaF, ολόκληρο το μήνυμα διασχίζει το κανάλι για να μεταφερθεί στον επόμενο κόμβο, και επομένως, θα χρειαστεί χρονο $(M + 1)t_w$. Επίσης, όλα τα flits του θα αποθηκευτούν στους buffers και θα διασχίσουν τον διαδρομητή, απαιτώντας συνολικό χρόνο διάσχισης $(M + 1)t_m$. Μετά την αποθήκευση, το flit επικεφαλίδας είναι το μόνο που χρησιμοποιείται για τη διαδρόμηση που απαιτεί χρόνο t_r. Η διαδικασία αυτή επαναλαμβάνεται D φορές και η όλη διαδικασία απαιτεί συνολικό χρόνο ίσο με:

$$\begin{aligned} T_{SaF} &= D[(M + 1)t_w + (M + 1)t_m + t_r] \\ &= D(t_w + t_r + t_m) + DM(t_w + t_m). \end{aligned}$$

Μεταγωγή VCT και wormhole Στη μεταγωγή virtual cut-through και wormhole, το flit επικεφαλίδας είναι αυτό που «χαράζει» τον δρόμο. Καθώς ολόκληρο το μήνυμα μεταφέρεται στο κανάλι, η είσοδος του flit της επικεφαλίδας σε έναν ενδιάμεσο κόμβο χρησιμοποιείται για τη λήψη απόφασης διαδρόμησης (χρόνος t_r), και στη συνέχεια διασχίζει τον διαδρομητή σε χρόνο t_m και εξέρχεται από το κανάλι εξόδου. Ουσιαστικά συμπεριφέρεται όπως και το probe στην μεταγωγή κυκλώματος, από τη στιγμή που υποθέτουμε ότι δεν βρίσκει κανένα εμπόδιο στο δρόμο του. Επομένως, αν δούμε μόνο το flit επικεφαλίδας, αυτό θα φτάσει στον προορισμό του σε χρόνο ακριβώς ίσο με $D(t_w + t_r + t_m)$. Τα υπόλοιπα flits ακολουθούν από πίσω, το ένα μετά το άλλο και χρησιμοποιούν τις ήδη σχηματισμένες συνδέσεις καναλιών εισόδου-εξόδου. Μετά από χρόνο Mt_w[1] έχουν καταφτάσει και τα υπόλοιπα M flits του μηνύματος. Επομένως,

$$T_{VCT} = T_{WS} = D(t_w + t_r + t_m) + Mt_w.$$

Συνοψίζοντας, οι παρακάτω σχέσεις μας δίνουν τον χρόνο που χρειάζεται ένα μήνυμα μήκους $M + 1$ flits να διασχίσει διαδρομή μήκους D με τους τέσσερις μηχανισμούς μεταγωγής:

$$T_{SaF} = D(t_w + t_r + t_m) + DM(t_w + t_m) \tag{3.3}$$
$$T_{CS} = D(t_w + t_r + t_m) + (M + 1)t_w \tag{3.4}$$
$$T_{VCT} = T_{WS} = D(t_w + t_r + t_m) + Mt_w. \tag{3.5}$$

Υπεραπλουστεύοντας, αν θεωρήσουμε ότι $t_w \approx t_r \approx t_m = 1$ χρονική μονάδα, βλέπουμε ότι οι απαιτούμενοι χρόνοι για τη μεταγωγή κυκλώματος, VCT και wormhole είναι τάξης $\Theta(D + M)$, ενώ για τη μεταγωγή SaF είναι τάξης $\Theta(DM)$. Υποθέτοντας επιπλέον ότι τα μηνύματα δεν είναι πολύ μικρά (δηλαδη ότι $M > D$ flits), τότε για τις τρεις πρώτες τεχνικές επικρατεί ο όρος M και ο απαιτούμενος χρόνος είναι τάξης $\approx \Theta(M)$. Το συμπέρασμα είναι πολύ σημαντικό: η μεταφορά μηνύματος με τις τεχνικές μεταγωγής κυκλώματος VCT και wormhole είναι εν πολλοίς *ανεξάρτητη της απόστασης*. Αντίθετα, η χρήση της μεταγωγής SaF απαιτεί χρόνο ευθέως ανάλογο της απόστασης που πρέπει να διανυθεί. Όλα αυτά βέβαια, με την προϋπόθεση ότι το μήνυμα δεν βρίσκει εμπόδια στο δρόμο του που το αναγκάζουν να υποστεί επιπλέον χρόνους αναμονής σε ενδιάμεσους κόμβους.

3.5.2 Σύγκριση τεχνικών μεταγωγής

Τα συμπεράσματα της προηγούμενης ενότητας έχουν σημαντικό αντίκτυπο στη σχεδίαση του δικτύου διασύνδεσης και εξηγούν και την ιστορική εξέλιξή του. Η τεχνική της μεταγωγής

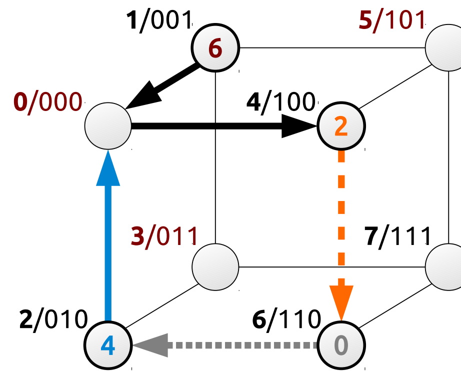
[1]Εδώ υποθέσαμε ότι $t_w \geq t_m$· σε αντίθετη περίπτωση τα υπόλοιπα flits φτάνουν ένα κάθε διάστημα t_m, οπότε γενικότερα ο χρόνος να φτάσουν όλα είναι $M \max\{t_w, t_r\}$. Επίσης, προσέξτε ότι αν οι διαδρομητές είχαν μόνο buffers εισόδου και όχι εξόδου, ο χρόνος θα είναι $M(t_w + t_r)$.

SaF ήταν αυτή που χρησιμοποιήθηκε σχεδόν αποκλειστικά στα πρώτα μεγάλα συστήματα κατανεμημένης μνήμης και ως τεχνική είναι γνωστή από τη δεκαετία του 1960, στηρίζοντας την ανάπτυξη των τοπικών δικτύων υπολογιστών αλλά και ολόκληρου του διαδικτύου. Η μεταγωγή wormhole, για παράδειγμα, είναι πολύ πιο πρόσφατη επινόηση (μέσα της δεκαετίας του 1980). Από τη στιγμή που η μεταγωγή SaF έχει χρόνους μεταφοράς ανάλογους της απόστασης, τα δίκτυα διασύνδεσης που επιλέγει ο σχεδιαστής θα πρέπει να έχουν μικρές αποστάσεις μεταξύ των κόμβων. Αυτός είναι και ο λόγος που τα πρώτα μεγάλα συστήματα κατανεμημένης μνήμης χρησιμοποιούσαν σχεδόν αποκλειστικά την τοπολογία του υπερκύβου παρά τα όσα μειονεκτήματα έχει. Από τον Πίνακα 3.1, ο υπερκύβος διαθέτει σημαντικά χαμηλότερη διάμετρο ($\log_2 N$ για N κόμβους) από π.χ. τη διάμετρο ένα διδιάστατου πλέγματος ίδιου μεγέθους ($\sqrt{N}$).

Η μεταφορά μηνυμάτων με τη χρήση των υπολοίπων τεχνικών, από την άλλη, δεν εξαρτάται σημαντικά από την απόσταση που πρέπει να διανυθεί. Χαλαρώνουν επομένως, οι απαιτήσεις για μικρές αποστάσεις μέσα στο δίκτυο διασύνδεσης και μπορούμε να χρησιμοποιήσουμε πιο εύκολα υλοποιήσιμα και κλιμακώσιμα δίκτυα, όπως πλέγματα και tori χαμηλής διάστασης παρά τις μεγάλες διαμέτρους τους. Είναι χαρακτηριστικό ότι από τα μέσα της δεκαετίας του 1990 που άρχισε να γίνεται ευρέως χρήση των τεχνικών VCT και wormhole μέχρι και σήμερα, ελάχιστα είναι τα συστήματα που δεν χρησιμοποιούν πλέγματα και tori.

Είναι σημαντικό όμως, να τονίσουμε ότι τόσο η παραπάνω ανάλυση όσο και το Σχ. 3.14, βασίζονται στην προϋπόθεση ότι το μεταφερόμενο μήνυμα δεν χρειάζεται να περιμένει σε κανέναν ενδιάμεσο κόμβο (δηλαδή όλα τα κανάλια που χρησιμοποιήθηκαν ήταν ελεύθερα), κάτι το οποίο δεν πρόκειται να ισχύει σε περιπτώσεις αυξημένης κίνησης στο δίκτυο. Σε τέτοιες περιπτώσεις, οι χρόνοι επικοινωνίας θα πρέπει να συμπεριλάβουν και τις καθυστερήσεις λόγω αναμονής στους ενδιάμεσους κόμβους (queueing delays). Η ανάλυση για αυτή την περίπτωση είναι συνήθως στοχαστική και δεν θα μας απασχολήσει εδώ. Είναι πάντως γεγονός ότι οι μεταγωγές κυκλώματος και wormhole ρίχνουν πολύ τις επιδόσεις τους σε συνθήκες μεγάλης κίνησης, καθώς δεσμεύουν πολλούς πολύτιμους πόρους του δικτύου (κανάλια). Αντίθετα, η μεταγωγή SAF μπορεί να απαιτεί χώρο για buffers, δεν δεσμεύει όμως κανάλια του δικτύου παρά μόνο προσωρινά, τη στιγμή που μεταφέρεται το μήνυμα σε γειτονικό κόμβο. Για τον λόγο αυτόν, μπορεί σε συνθήκες υψηλής κίνησης να έχει καλύτερη συμπεριφορά από τις άλλες τεχνικές. Τέλος, προσέξτε ότι η μεταγωγή VCT διαθέτει την ταχύτητα των μεταγωγών κυκλώματος και wormhole εφόσον οι συνθήκες είναι ευνοϊκές, συμπεριφέρεται όμως ως μεταγωγή SAF σε περίπτωση αυξημένης κίνησης.

Ένα άλλο σοβαρό ζήτημα είναι το πρόβλημα του *αδιεξόδου* (deadlock). Το πρόβλημα αυτό μπορεί να εμφανιστεί όταν σε κάποια δεδομένη χρονική στιγμή στο δίκτυο επικοινωνούν ταυτόχρονα δύο ή παραπάνω ζευγάρια κόμβων και άρα, σχηματίζονται, την ίδια στιγμή, πολλαπλές διαδρομές. Αν δεν έχει υπάρξει ειδική πρόβλεψη, τότε μπορεί να δημιουργηθεί μία κατάσταση στην οποία καμία από τις διαδρομές δεν μπορεί να συνεχι-

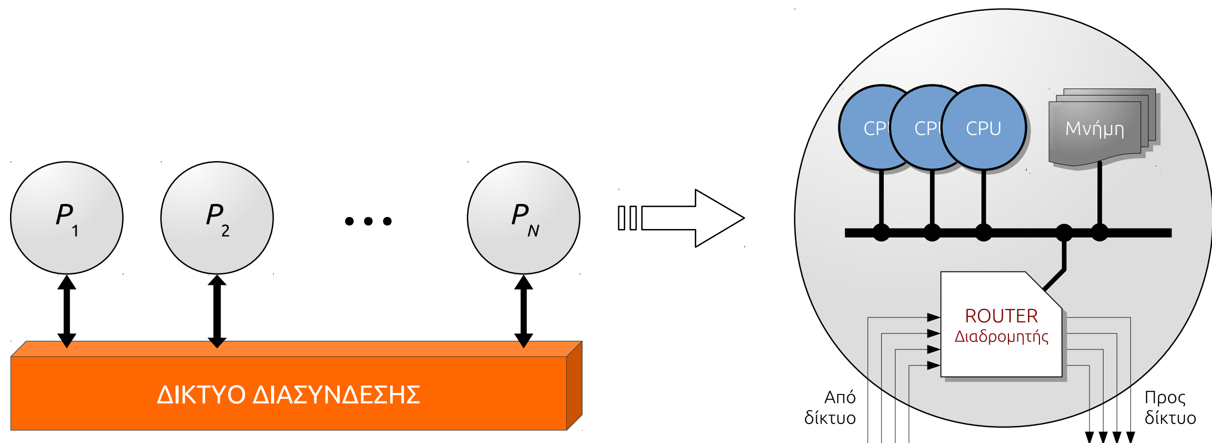


Σχήμα 3.16 *Τέσσερις ταυτόχρονες διαδρομές προκαλούν αδιέξοδο σε 3-κύβο με μεταγωγή wormhole ή κυκλώματος: 1→6, 2→4, 4→2 και 6→0.*

στεί, διότι την εμποδίζει κάποια από τις υπόλοιπες και επομένως, οι κόμβοι περιμένουν ατελείωτα να ολοκληρωθεί η επικοινωνία τους. Αν και το πρόβλημα του αδιεξόδου μπορεί να εμφανιστεί με όλα τα είδη μεταγωγής, οι μεταγωγές κυκλώματος και wormhole είναι εξαιρετικά επιρρεπείς σε αυτό. Ο λόγος είναι ότι δεσμεύουν πολλά κανάλια του δικτύου μέχρι να γίνει η μεταφορά του μηνύματος. Ένα παράδειγμα δίνεται στο Σχ. 3.16 σε έναν τριδιάστατο υπερκύβο, όπου ο κάθε κόμβος που στέλνει μήνυμα έχει σημειωμένο τον προορισμό του. Υποθέτοντας μεταγωγή κυκλώματος ή wormhole, οι τέσσερις διαδρομές έχουν προχωρήσει μερικώς, και πλέον, η μία εμποδίζει την άλλη να ολοκληρωθεί.

Υπάρχουν αρκετές στρατηγικές προκειμένου να δοθεί λύση στο πρόβλημα του αδιεξόδου. Μία από αυτές είναι η χρήση κατάλληλης διαδρόμησης η οποία αποφεύγει παντελώς το πρόβλημα. Για παράδειγμα, μπορεί να αποδειχτεί ότι με τη διαδρόμηση e-cube, που είδαμε στην Ενότητα 3.4.1, δεν εμφανίζεται ποτέ αδιέξοδο. Όμως, δεν υπάρχουν πάντα τέτοιοι αλγόριθμοι διαδρόμησης σε όλες τις τοπολογίες. Η διαφορετική τεχνική που χρησιμοποιείται σε συστήματα με μεταγωγή wormhole για τη μείωση της πιθανότητας αδιεξόδων, είναι τα λεγόμενα *εικονικά κανάλια* (virtual channels), τα οποία ουσιαστικά κάνουν το ένα φυσικό κανάλι που υπάρχει μεταξύ δύο κόμβων να συμπεριφέρεται σαν πολλά διαφορετικά κανάλια, αυξάνοντας έτσι τις δυνατότητες για διαδρόμηση χωρίς αδιέξοδα. Ένα φυσικό κανάλι «διαιρείται» σε πολλά εικονικά κανάλια τα οποία μοιράζονται *χρονικά* το φυσικό κανάλι. Κάθε εικονικό κανάλι απαιτεί δικό του buffer σε κάθε κόμβο ο οποίος χρησιμοποιείται για να αποθηκεύσει ένα flit μέχρι να έρθει η ώρα να χρησιμοποιήσει το φυσικό κανάλι. Γι' αυτόν τον λόγο, προφανώς, η τεχνική δεν μπορεί να χρησιμοποιηθεί στη μεταγωγή κυκλώματος. Πάντως, η προσθήκη εικονικών καναλιών αυξάνει σημαντικά την πολυπλοκότητα και επομένως, το κόστος ενός διαδρομητή.

Τέλος, η χρήση της μεταγωγής wormhole αποκλείει εκ φύσεως τη χρήση τεχνικών οπισθοχώρησης (backtracking). Η ισχυρή αυτή τεχνική χρησιμοποιείται συχνά σε δίκτυα με μεταγωγή κυκλώματος ως εξής: αν μία διαδρομή σταματήσει για αρκετή ώρα σε έναν ενδιάμεσο κόμβο λόγω κίνησης, τότε, προκειμένου να απελευθερωθούν πόροι του δικτύου,



Σχήμα 3.17 *Οργάνωση ομαδοποιημένων πολυεπεξεργαστών.*

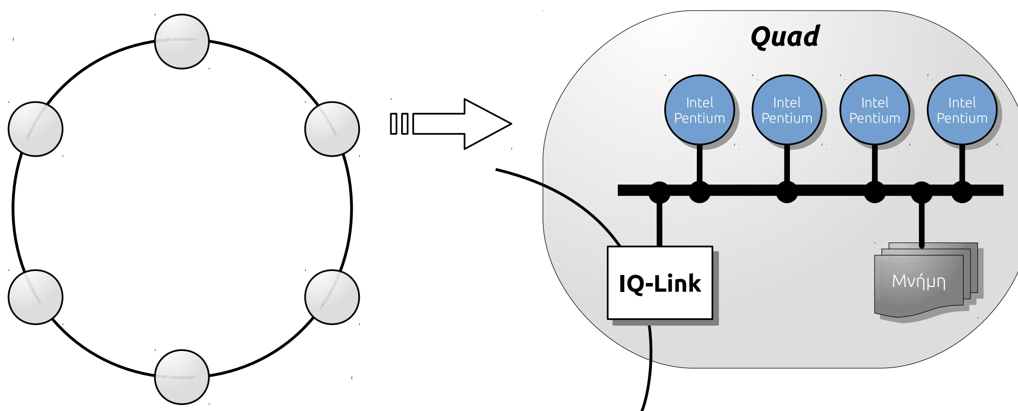
το διερευνητικό σήμα σχηματισμού κυκλώματος οπισθοχωρεί σε έναν προηγούμενο κόμβο, απελευθερώνοντας τα ενδιάμεσα κανάλια που είχε δεσμεύσει μέχρι εκείνη τη στιγμή. Στη συνέχεια, ξαναπροσπαθεί κάποια στιγμή αργότερα, ακολουθώντας την ίδια διαδρομή ή κάποια άλλη, εφόσον η διαδρόμηση είναι προσαρμοζόμενη.

3.6 Ομαδοποιημένοι πολυεπεξεργαστές και υπολογιστικές συστάδες

Η πιο σύγχρονη εκδοχή των πολυεπεξεργαστών κατανεμημένης μνήμης είναι οι *ομαδοποιημένοι πολυεπεξεργαστές* οργάνωση η οποία φαίνεται στο Σχ. 3.17. Η αρχιτεκτονική τους αποτελεί συνδυασμό κοινόχρηστης και κατανεμημένης μνήμης. Η μνήμη συνεχίζει να είναι κατανεμημένη ανάμεσα στους κόμβους, οι οποίοι ενώνονται με το δίκτυο διασύνδεσης. Ο κάθε κόμβος, όμως, είναι πιο σύνθετος και αντί για έναν επεξεργαστή, εμπεριέχει μία ολόκληρη ομάδα από επεξεργαστές, η οποία αποτελεί συνήθως έναν μικρό συμμετρικό πολυεπεξεργαστή. Οι επεξεργαστές της ομάδας μοιράζονται μεταξύ τους τον δίαυλο και την τοπική μνήμη, αλλά και τον διαδρομητή μέσω του οποίου επικοινωνούν με τους υπόλοιπους κόμβους.

Χαρακτηριστικό παράδειγμα αποτελεί ο SGI Origin 2000 στον οποίο η κάθε ομάδα διέθετε δύο επεξεργαστές με κοινόχρηστη μνήμη, τοποθετημένους επάνω σε κοινό δίαυλο. Η ομάδα αποτελούσε μία πλακέτα-κόμβο του συστήματος. Οι κόμβοι αυτοί συνδέονταν μεταξύ τους με ένα δίκτυο υπερκύβου (για παραπάνω από 32 πλακέτες-κόμβους η τοπολογία τροποποιείται ελαφρά). Παρόμοια οργάνωση χρησιμοποιήθηκε και σε επόμενα συστήματα της SGI, όπως τα UV-1000 και UV-2000.

Ένα άλλο παράδειγμα, είναι ο Numa-Q της Sequent, η χονδρική δομή του οποίου φαίνεται στο Σχ. 3.18. Στον υπολογιστή αυτό, η κάθε ομάδα («quad») είναι ένας συμμετρι-



Σχήμα 3.18 *Χονδρική δομή του Sequent Numa-Q.*

κός πολυεπεξεργαστής με 4 επεξεργαστές Intel Pentium Pro. Οι ομάδες έχουν διασύνδεση δακτυλίου. Την επικοινωνία μεταξύ διαύλου και δακτυλίου σε κάθε ομάδα την αναλαμβάνει μία ειδική μονάδα («IQ-Link» στο σχήμα).

Στις μέρες μας που η επικράτηση των επεξεργαστών πολλαπλών πυρήνων είναι καθολική, μια παραλλαγή της οργάνωσης του Σχ. 3.17 είναι αυτή που συναντούμε πιο συχνά στην πράξη. Οι κόμβοι του δικτύου δεν αποτελούνται από ομάδες πολυεπεξεργαστών κοινόχρηστης μνήμης, αλλά από έναν (γενικά λίγους) επεξεργαστή, συνήθως 2-8 πυρήνων.

3.6.1 Υπολογιστικές συστάδες

Οι μεγάλοι πολυεπεξεργαστές κατανεμημένης μνήμης, που όπως είπαμε είναι γνωστοί και ως μαζικά παράλληλοι (MPP), αποτελούσαν ανέκαθεν υπερυπολογιστικά συστήματα κορυφαίων επιδόσεων, αλλά και τεράστιου οικονομικού κόστους. Ενδεικτικά αναφέρουμε ότι το σύστημα Earth Simulator, που έφτιαξε η NEC για συγκεκριμένα Ιαπωνικά ερευνητικά κέντρα και το οποίο αποτέλεσε το ταχύτερο σύστημα στον κόσμο μεταξύ 2002 και 2004, κόστισε $400.000.000. Παρότι ο Earth Simulator είναι, ίσως, ο ακριβότερος υπολογιστής που φτιάχτηκε ποτέ, οι κορυφαίες θέσεις της λίστας Top500 με τα ισχυρότερα υπολογιστικά συστήματα στον πλανήτη, καταλαμβάνονται από μηχανές με κόστος άνω των $100.000.000 το καθένα. Ακόμα και «ταπεινότεροι» υπολογιστές με λίγες εκατοντάδες κόμβους, είχαν κόστη που ξεπερνούσαν το ένα εκατομμύριο δολάρια. Όλα αυτά τα συστήματα ήταν κατασκευασμένα για στοχευμένες υπολογιστικές ανάγκες και όχι για γενική και ευρεία χρήση. Η στόχευση για υψηλές επιδόσεις απαιτεί και εξειδικευμένες αρχιτεκτονικές σε επίπεδο επεξεργαστή, υποσυστήματος μνήμης και, ακόμα περισσότερο, δικτύου διασύνδεσης. Αν επιπλέον, προσμετρήσουμε και το κόστος υλοποίησης εξειδικευμένων λειτουργικών συστημάτων, εργαλείων και υποδομής ανάπτυξης λογισμικού, μπορούμε να καταλάβουμε γιατί τα συστήματα αυτά είναι εξωπραγματικά ακριβά.

Οι υπολογιστικές *συστάδες* (clusters) και *πλέγματα* (grids) αποτελούν τον αντίποδα

των MPP. Η σχεδίασή τους στηρίζεται στη χρήση οικονομικού και ευρέως διαθέσιμου υλικού (επεξεργαστών, μνημών και δικτυακής υποδομής) και λογισμικού συστήματος (λειτουργικά συστήματα, μεταφραστές, βιβλιοθήκες κλπ). Πιο συγκεκριμένα, στην βασική τους μορφή οι συστάδες είναι απλά ένα σύνολο από προσωπικούς υπολογιστές PC που συνδέονται μεταξύ τους με ένα συμβατικό τοπικό δίκτυο. Συνδυάζοντας αρκετά τυποποιημένα PC με απλές κάρτες δικτύωσης τύπου ethernet, και εξοπλίζοντάς τα με λογισμικό συστήματος σχεδόν μηδενικού κόστους, όπως π.χ. λειτουργικά συστήματα ανοιχτού κώδικα σαν το Linux, μπορεί κανείς να έχει ένα αξιόλογο σύστημα κατανεμημένης μνήμης, χωρίς βέβαια να περιμένει κορυφαίες επιδόσεις.

Παρά την αυτονομία των κόμβων, οι συστάδες είναι συνήθως ομοιογενείς, αποτελούμενες από όμοιους υπολογιστές, τοποθετημένους στον ίδιο χώρο, οι οποίοι διαθέτουν ίδια λειτουργικά συστήματα—ως επί το πλείστον τύπου Linux. Τα υπολογιστικά πλέγματα από την άλλη, διαφοροποιούνται στο ότι μπορεί να περιέχουν ανομοιογενείς κόμβους με διαφορετικά λειτουργικά συστήματα και να είναι γεωγραφικά εξαπλωμένα σε ευρύτερες περιοχές. Στη συνέχεια, θα εστιάσουμε στις συστάδες, αν και όσα συζητήσουμε ισχύουν και για τα υπολογιστικά πλέγματα.

Εκτός από τη χρήση ευρέως διαθέσιμου υλικού, υπάρχουν πολλές άλλες διαφορές μεταξύ συστάδων και μαζικά παράλληλων συστημάτων κατανεμημένης μνήμης.

- Οι συστάδες είναι μία συλλογή από αυτόνομους, ανεξάρτητους, ολοκληρωμένους υπολογιστές και όχι ένα ενιαίο σύστημα.
- Ο κάθε κόμβος σε μία συστάδα έχει το δικό του λειτουργικό σύστημα και χώρο διευθύνσεων μνήμης.
- Η κάρτα δικτύου παίζει τον ρόλο του διαδρομητή αλλά τουλάχιστον, οι κάρτες για τα συνήθη δίκτυα (π.χ. ethernet) ενσωματώνουν εντελώς διαφορετική λογική:
  - Απευθύνονται σε γενικά δίκτυα τα οποία δεν έχουν γνωστή εκ των προτέρων δομή, σε αντίθεση με τους MPP στους οποίους η τοπολογία του δικτύου διασύνδεσης είναι αναπόσπαστο τμήμα της αρχιτεκτονικής τους.
  - Η τοπολογία του δικτύου σε μία συστάδα μπορεί να είναι σχετικά άναρχη και να αλλάζει εύκολα με απλή προσθήκη / αφαίρεση υπολογιστών.
  - Λόγω των παραπάνω, οι αλγόριθμοι για διαδρόμηση και μεταφορά μηνυμάτων είναι γενικοί και κυρίως αργοί.

Παρόλο που μία συστάδα αποτελείται από αυτόνομους υπολογιστές, μπορεί να προγραμματιστεί με τρόπο που να φαίνεται ως μία παράλληλη μηχανή. Πολλές συστάδες διαθέτουν επίσης, ενδιάμεσο λογισμικό το οποίο τοποθετείται ανάμεσα στο λειτουργικό

σύστημα και τα προγραμματιστικά εργαλεία και δίνει την εικόνα ενός *ενιαίου συστήματος* (single system image, SSI) στους χρήστες, τους προγραμματιστές και τους διαχειριστές. Για παράδειγμα, η συστάδα μπορεί να φαίνεται στους χρήστες ως ένας απλός υπολογιστής στον οποίο μπορούν να συνδεθούν και να υποβάλλουν την εφαρμογή τους για εκτέλεση, χωρίς να γνωρίζουν τους υπόλοιπους κόμβους. Κατά τη διάρκεια της εκτέλεσης, το ενδιάμεσο λογισμικό μπορεί να μεταφέρει διαφανώς διεργασίες μεταξύ των υπολογιστών της, ώστε να εξασφαλίζει καλύτερη διαχείριση πόρων.

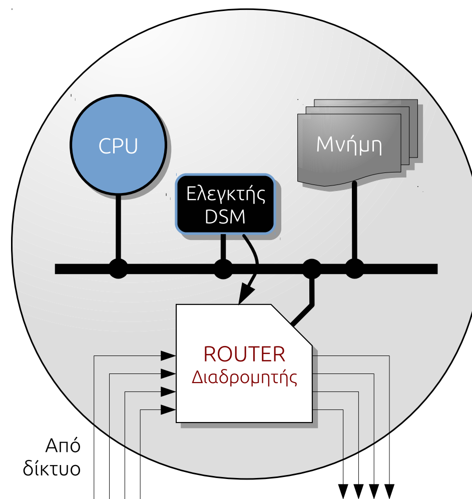
Το σημαντικό πρόβλημα του δικτύου χαμηλών επιδόσεων λύνεται σε μεγάλο βαθμό με τη χρήση ταχύτερων δικτύων και αποφυγή των κλασικών δικτυακών πρωτοκόλλων επικοινωνίας, με χρήση «ελαφριών», πιο αποδοτικών πρωτοκόλλων (π.χ. active messages). Ταχύτερα δίκτυα, όπως για παράδειγμα το Myrinet και το Infiniband, παρέχουν (με το ανάλογο κόστος) επικοινωνίες πολύ χαμηλής καθυστέρησης που φέρνουν ένα τοπικό δίκτυο συστάδας να συναγωνίζεται ακόμα και τα εξειδικευμένα δίκτυα διασύνδεσης των MPP.

Οι υπολογιστικές συστάδες έχουν εξελιχθεί σε απίστευτο βαθμό. Από απλά, οικονομικά, μικρά έως μεσαία συστήματα, έχουν φτάσει να αποτελούν τους κορυφαίους, σύγχρονους υπερυπολογιστές στον κόσμο, εκτοπίζοντας ακόμα και τους μεγαλύτερους MPP σε επιδόσεις. Οι υπολογιστές-κόμβοι αποτελούνται σχεδόν πάντα από πολυπύρηνους επεξεργαστές, οπότε ουσιαστικά το όλο σύστημα διαθέτει οργάνωση ομαδοποιημένων πολυεπεξεργαστών (Σχ. 3.17). Είναι σημαντικό, πάντως, να σημειωθεί ότι σε όλες αυτές τις εξελίξεις η ύπαρξη λογισμικού ανοιχτού κώδικα και εξαιρετικής ποιότητας έπαιξε καταλυτικό ρόλο.

3.7 Κατανεμημένη κοινή μνήμη

Στην ενότητα αυτή θα ασχοληθούμε με τα συστήματα τα οποία υποστηρίζουν τη λεγόμενη οργάνωση κατανεμημένης κοινής μνήμης. Πρόκειται κατά βάση για μηχανές στις οποίες η μνήμη είναι κατανεμημένη, αλλά ο χειρισμός της γίνεται διαφανώς από το υλικό, σαν να είναι κοινόχρηστη για όλους τους επεξεργαστές.

Οι μηχανές *κατανεμημένης κοινής μνήμης* (distributed shared memory, DSM) είναι πολυεπεξεργαστές κατανεμημένης μνήμης, οι οποίοι όμως διαθέτουν επιπλέον υλικό σε κάθε κόμβο που δίνει την εντύπωση στον επεξεργαστή ότι συμμετέχει σε μηχανή κοινόχρηστης μνήμης. Συγκεκριμένα, υπάρχει ένας λογικά ενιαίος χώρος διευθύνσεων ο οποίος, όμως, είναι στην πράξη διαχωρισμένος και η μνήμη του κάθε κόμβου αποτελεί ένα τμήμα του. Οι αιτήσεις του επεξεργαστή (ο οποίος απευθύνεται στον ενιαίο χώρο—όπως στις μηχανές κοινόχρηστης μνήμης) παρακολουθούνται από ειδικό ελεγκτή (DSM controller), όπως φαίνεται στο Σχ. 3.19 (ο ελεγκτής αυτός αποτελεί συχνά ενιαία συσκευή με τον διαδρομητή). Αν ο ελεγκτής διαπιστώσει ότι το δεδομένο που ζητά ο επεξεργαστής βρίσκεται στην



Σχήμα 3.19 *Κόμβος σε σύστημα* DSM.

τοπική του μνήμη, επιτρέπει σε αυτή να το δώσει αμέσως. Αν όμως, το δεδομένο βρίσκεται σε κάποιον άλλο κόμβο, δημιουργείται (εν αγνοία του επεξεργαστή) ένα μήνυμα και αποστέλλεται μέσω του διαδρομητή στον κόμβο που περιέχει το δεδομένο, ζητώντας το. Μόλις παραληφθεί το δεδομένο, προωθείται στον επεξεργαστή ο οποίος απλά παρατηρεί μία αργοπορία στην παράδοση. Με αυτόν τον τρόπο, ακόμα και από προγραμματιστικής πλευράς, δίνεται η εντύπωση μίας αρχιτεκτονικής με κοινόχρηστη μνήμη. Η μνήμη είναι *λογικά* κοινόχρηστη αλλά *φυσικά* κατανεμημένη.

Λόγω της διαφοράς στην ταχύτητα προσπέλασης των τοπικών και των απομακρυσμένων δεδομένων που αντιλαμβάνεται ο επεξεργαστής, οι μηχανές αυτές είναι γνωστές και ως μηχανές *ανομοιόμορφης προσπέλασης μνήμης* (non-uniform memory access, NUMA). Για παράδειγμα, ο Cray T3D βασιζόταν σε τριδιάστατο torus και υποστήριζε κατανεμημένη κοινή μνήμη· οι τοπικές προσπελάσεις απαιτούσαν 2 κύκλους, ενώ οι απομακρυσμένες απαιτούσαν περίπου 150 κύκλους ρολογιού.

Προκειμένου να μειωθεί ο χρόνος προσπέλασης των μη τοπικών δεδομένων, είναι φυσιολογική και, ίσως, επιβεβλημένη η χρήση κρυφών μνημών για τα δεδομένα αυτά σε κάθε κόμβο. Όμως, από τη στιγμή που εισάγονται κρυφές μνήμες, θα πρέπει κανείς να προσέξει το γνωστό μας, από το Κεφάλαιο 2, πρόβλημα της συνοχής. Έτσι, οι μηχανές που χρησιμοποιούν κρυφές μνήμες, διαθέτουν συνήθως και κάποιο πρωτόκολλο που εξασφαλίζει τη συνοχή των δεδομένων (μηχανές cache coherent NUMA ή CCNUMA). Λόγω της έλλειψης κοινού μέσου παρακολούθησης, είναι απαραίτητη η χρήση πρωτοκόλλων με καταλόγους. Ο ερευνητικός υπολογιστής DASH του Πανεπιστημίου Stanford ήταν το πρώτο σύστημα με κατανεμημένη κοινή μνήμη που έκανε χρήση πρωτοκόλλου συνοχής με καταλόγους.

Φυσικά, είναι πιθανές και οι υπόλοιπες στρατηγικές που αποσκοπούν στην αποφυγή του προβλήματος, τις οποίες συζητήσαμε στο Κεφάλαιο 2. Για παράδειγμα, στον Cray T3D δεν χρησιμοποιήθηκε κάποιο πρωτόκολλο συνοχής—απλά τα κοινόχρηστα δεδομένα δεν

επιτρεπόταν να αντιγραφούν στις κρυφές μνήμες. Έτσι, κάθε απομακρυσμένο κοινόχρηστο δεδομένο θα έπρεπε να μεταφέρεται μέσω του δικτύου κάθε φορά στον επεξεργαστή που το ζήτησε, απαιτώντας μονίμως αρκετούς κύκλους ρολογιού για να παραδοθεί.

Η οργάνωση κατανεμημένης κοινής μνήμης είναι δυνατή και σε συστήματα που είναι οργανωμένα ως ομαδοποιημένοι πολυεπεξεργαστές. Σε αυτή την περίπτωση, ο ελεγκτής DSM παρακολουθεί τις κινήσεις στον δίαυλο που προκαλούνται από όλους τους τοπικούς επεξεργαστές και ενεργεί ακριβώς όπως περιγράφτηκε παραπάνω. Οι ομαδοποιημένοι πολυεπεξεργαστές ξεκίνησαν από την ανάγκη να υλοποιήσει κανείς συστήματα κοινόχρηστης μνήμης τα οποία, όμως, μπορούν να επιδέχονται αναβαθμίσεις (αύξηση του αριθμού των επεξεργαστών) με ανάλογη κλιμάκωση των επιδόσεών τους. Κάτι τέτοιο, όπως είπαμε στο Κεφάλαιο 2, δεν μπορεί να γίνει με βάση έναν απλό δίαυλο. Η λύση στο πρόβλημα είναι να χρησιμοποιηθούν μικρά και δοκιμασμένα συστήματα, βασισμένα σε δίαυλο (SMP)· με την προσθήκη ενός κοινόχρηστου διαδρομητή και ενός ελεγκτή DSM, ένα μικρό σύστημα SMP μπορεί να αποτελέσει τον δομικό λίθο για πολύ μεγαλύτερα συστήματα, τα οποία ολοκληρώνονται μέσω ενός δικτύου διασύνδεσης. Για τον λόγο αυτόν και τα συγκεκριμένα συστήματα είναι επίσης γνωστά ως *κλιμακώσιμοι πολυεπεξεργαστές* (scalable multiprocessors).

Βέβαια, όπως ήδη είδαμε, η υποστήριξη κοινής μνήμης σε όλο το πλάτος του πολυεπεξεργαστή σημαίνει και εύρεση λύσης στο πρόβλημα της συνοχής. Δεν φτάνει μόνο να υπάρχει συνοχή στις κρυφές μνήμες εντός μίας ομάδας—θα πρέπει να υπάρχει συνοχή και μεταξύ των ομάδων. Έτσι, οι ομαδοποιημένοι επεξεργαστές διαθέτουν πολλές φορές δύο πρωτόκολλα συνοχής· ένα για τους συμμετρικούς πολυεπεξεργαστές της κάθε ομάδας (το «εσωτερικό» πρωτόκολλο) και ένα για την διατήρηση συνοχής μεταξύ των ομάδων (το «εξωτερικό» πρωτόκολλο).

Εσωτερικά, χρησιμοποιείται συνήθως κάποιο πρωτόκολλο παρακολούθησης λόγω της ύπαρξης του διαύλου. Εξωτερικά, απαιτείται πρωτόκολλο με καταλόγους. Στον Numa-Q (Σχ. 3.18) ως πρωτόκολλο μεταξύ των ομάδων χρησιμοποιείται το αλυσιδωτό πρωτόκολλο IEEE SCI. Το πρωτόκολλο το υλοποιεί για όλη την ομάδα το IQ-Link, το οποίο διαθέτει την απαραίτητη κρυφή μνήμη για τα απομακρυσμένα δεδομένα (32 Mbytes) μαζί με τις πληροφορίες καταλόγου. Έτσι, για το εξωτερικό πρωτόκολλο η ομάδα δείχνει ως μία οντότητα. Εσωτερικά στην ομάδα, τα τοπικά δεδομένα αλλά και τα μη τοπικά αντίγραφα στην κρυφή μνήμη του IQ-Link διατηρούν τη συνοχή τους μέσω πρωτοκόλλου παρακολούθησης MESI.

Ο προσεκτικός αναγνώστης θα πρέπει να έχει παρατηρήσει ότι η οργάνωση κατανεμημένης κοινής μνήμης δεν αφορά τις υπολογιστικές συστάδες. Είναι ουσιαστικά ανεφάρμοστη σε αυτές, καθώς οι συμμετέχοντες υπολογιστές έχουν εντελώς αυτόνομους επεξεργαστές και χώρους διευθύνσεων μνήμης. Οποιαδήποτε προσπάθεια υποστήριξης DSM, θα απαιτούσε τόσο αλλαγές στο υλικό των υπολογιστών αυτών (κάτι που ξεφεύγει από τη βασική ιδέα του οικονομικού και ευρέως χρησιμοποιούμενου υλικού) όσο και στον πυρήνα των λειτουργικών συστημάτων τους, ώστε με κάποιο τρόπο να ενοποιηθούν οι ανεξάρτητοι

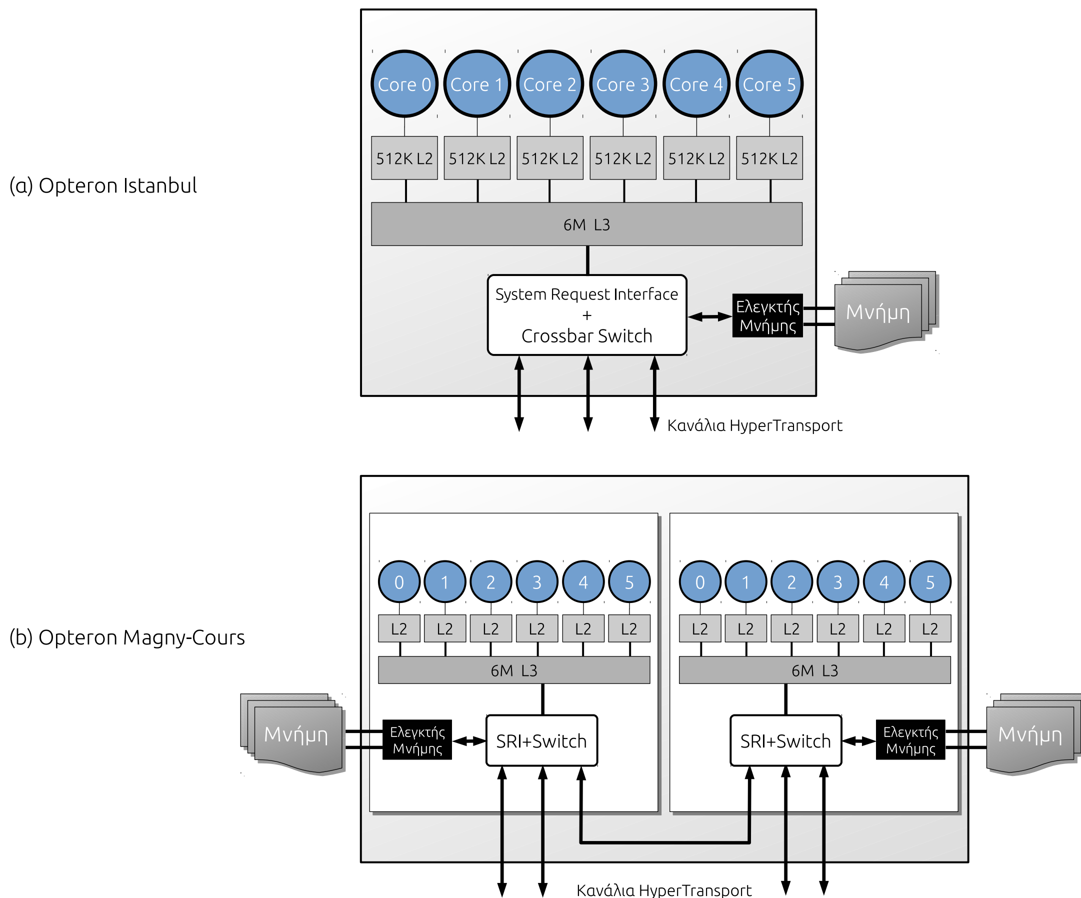
χώροι μνήμης. Όμως, η αλήθεια είναι ότι *μπορεί* να υποστηριχθεί κοινόχρηστη μνήμη, και μάλιστα χωρίς αλλαγές στο υλικό ή τον πυρήνα του λειτουργικού συστήματος. Η λειτουργικότητα αυτή υλοποιείται καθαρά σε λογισμικό με ειδικές βιβλιοθήκες, οι οποίες εκτελούνται σε επίπεδο χρήστη και είναι γνωστές ως κατανεμημένη κοινή μνήμη λογισμικού (software DSM, SDSM) ή κοινόχρηστη εικονική μνήμη (shared virtual memory, SVM). Ο περιορισμός είναι ότι μόνο σε επίπεδο εφαρμογών μπορεί να γίνει χρήση των βιβλιοθηκών αυτών και όχι σε επίπεδο υπηρεσιών του πυρήνα. Επίσης, οι επιδόσεις των εφαρμογών που κάνουν χρήση SDSM είναι εν γένει κατώτερες των προσδοκιών, λόγω της υψηλής κίνησης μηνυμάτων που δημιουργούν οι βιβλιοθήκες αυτές προκειμένου να παρέχουν την ψευδαίσθηση της κοινόχρηστης μνήμης.

3.8 Πολυπύρηνοι επεξεργαστές και κατανεμημένη μνήμη

Στην Ενότητα 2.8, είδαμε ότι οι σύγχρονοι πολυπύρηνοι επεξεργαστές με σχετικά μικρό αριθμό πυρήνων είναι οργανωμένοι ως μικροί συμμετρικοί πολυεπεξεργαστές με πιο πολύπλοκες, όμως, ιεραρχίες μνήμης. Όπως και στους συμμετρικούς πολυεπεξεργαστές, όταν ο αριθμός των πυρήνων μεγαλώνει, οι επιδόσεις της κοινόχρηστης μνήμης μειώνονται. Είναι λογικό λοιπόν ότι αργά ή γρήγορα θα πρέπει κανείς να καταφύγει στην οργάνωση κατανεμημένης μνήμης προκειμένου να υποστηριχτεί αποδοτική κλιμάκωση του πλήθους των πυρήνων. Μπορούμε να διακρίνουμε δύο περιπτώσεις για πολυπύρηνους επεξεργαστές που χρησιμοποιούνται σε συστήματα κατανεμημένης μνήμης:

- ολιγοπύρηνοι επεξεργαστές που μπορούν να χρησιμοποιηθούν ως κόμβοι σε ένα μεγαλύτερο σύστημα,
- επεξεργαστές με πολλούς έως πάρα πολλούς πυρήνες, οι οποίοι αποτελούν από μόνοι τους ολόκληρο σχεδόν το παράλληλο σύστημα, ενσωματώνοντας πυρήνες, μνήμες και δίκτυο διασύνδεσης σε ένα ολοκληρωμένο σύστημα (γνωστοί και ως multiprocessor system-on-chip, MPSoC).

Στην πρώτη κατηγορία, οι επεξεργαστές παρέχουν πολλαπλούς πυρήνες, εξοπλισμένους με επιπλέον υλικό το οποίο μπορεί να χρησιμοποιηθεί για επικοινωνία με άλλους επεξεργαστές. Ουσιαστικά το υλικό αυτό υλοποιεί τον διαδρομητή και πολλές φορές και τον ελεγκτή DSM, ώστε να υποστηριχτεί κοινός χώρος διευθύνσεων σε ολόκληρο το σύστημα. Χαρακτηριστικά παραδείγματα αποτελούν οι επεξεργαστές Opteron της AMD και Xeon (σειρές E5 και E7) της Intel. Και οι δύο οικογένειες διαθέτουν κανάλια για σύνδεση τοπικής μνήμης η οποία είναι κοινόχρηστη μεταξύ των πυρήνων του επεξεργαστή, αλλά και επιπλέον κανάλια για σχηματισμό δικτύου διασύνδεσης. Ένας τέτοιος επεξεργαστής αποτελεί, δηλαδή, ολοκληρωμένο κόμβο-ομάδα, και σύνολο από διασυνδεδεμένους επε-



Σχήμα 3.20 *Οργάνωση πολυπύρηνων επεξεργαστών Opteron.*

ξεργαστές σχηματίζει έναν ομαδοποιημένο πολυεπεξεργαστή (Σχ. 3.17). Τέλος, και οι δύο υποστηρίζουν κατανεμημένη κοινή μνήμη.

Στο Σχ. 3.20(α) φαίνεται η οργάνωση του 6-πύρηνου επεξεργαστή Opteron με κωδική ονομασία «Istanbul». Πρόκειται για ένα μικρό 6-πύρηνο σύστημα κοινόχρηστης μνήμης, όπου οι πυρήνες μοιράζονται την κρυφή μνήμη τρίτου επιπέδου. Επιπλέον, συμπεριλαμβάνεται διαδρομητής που παρέχει 3 κανάλια σύνδεσης προς παρόμοιους επεξεργαστές ή άλλες συσκευές εισόδου / εξόδου. Χωρίς τη χρήση επιπλέον υλικού υποστηρίζονται μέχρι 8 επεξεργαστές συνολικά. Τα κανάλια ονομάζονται HyperTransport και υλοποιούν δίκτυο με μεταγωγή πακέτου, αν και υπάρχουν υλοποιήσεις του HyperTransport με μεταγωγή cut-through.

Στο Σχ. 3.20(b) απεικονίζεται η δομή του 12-πύρηνου επεξεργαστή Opteron «Magny-Cours». Ο επεξεργαστής ουσιαστικά αποτελείται από δύο 6-πύρηνους Istanbul τοποθετημένους σε ένα ολοκληρωμένο κύκλωμα και ενωμένους ως *γείτονες*, δεσμεύοντας ένα κανάλι από τον καθένα. Επομένως, πρόκειται για ένα ζεύγος ανεξάρτητων κόμβων μέσα στον ίδιο επεξεργαστή. Τα 4 συνολικά κανάλια που απομένουν μπορούν να χρησιμοποιηθούν για το

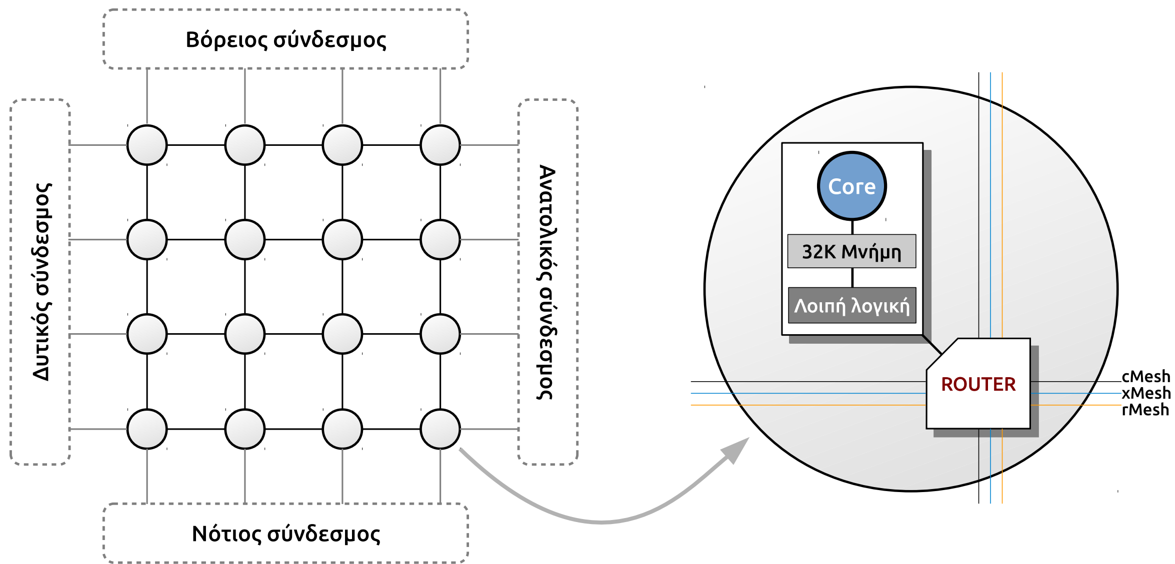
δίκτυο διασύνδεσης σε συστήματα μέχρι 4 επεξεργαστών και επομένως, μέχρι 48 πυρήνες υποστηρίζονται συνολικά (πάντα, χωρίς τη χρήση επιπλέον υλικού).

Επειδή το HyperTransport έχει υλοποιηθεί με πολύ χαμηλή καθυστέρηση, ο λεγόμενος *παράγοντας* NUMA (NUMA factor), δηλαδή το πόσο αργή είναι η προσπέλαση ενός δεδομένου από μία απομακρυσμένη μνήμη σε σχέση με την τοπική, είναι σχετικά χαμηλός. Συγκεκριμένα, είναι περίπου 1,2 (20% πιο αργά) για προσπέλαση γειτονικής μνήμης και φτάνει μέχρι 1,5 για προσπέλαση μνήμης σε απόσταση 2.

Οι επεξεργαστές Opteron αποτελούν κόμβους συστημάτων CCNUMA, καθώς η κρυφή μνήμη τρίτου επιπέδου μπορεί να αποθηκεύσει απομακρυσμένα δεδομένα και ο ελεγκτής μνήμης υλοποιεί πρωτόκολλο συνοχής για τα δεδομένα αυτά. Επειδή όμως, προορίζονται για συστήματα με σχετικά μικρό αριθμό επεξεργαστών, η συνοχή επιτυγχάνεται με έναν μάλλον «πρωτόγονο» τρόπο. Συγκεκριμένα, δεν χρησιμοποιούνται κατάλογοι και επομένως, λόγω άγνοιας του ποιοι επεξεργαστές έχουν αντίγραφα, σε κάθε τροποποίηση δεδομένου απαιτείται χρονοβόρα συνεννόηση μεταξύ *όλων των κόμβων* του συστήματος. Από τη σειρά Istanbul και μετά, υπάρχει η βελτίωση που ονομάζεται HyperTransport Assist, όπου 1MiB της κρυφής μνήμης 3ου επιπέδου δεσμεύεται προκειμένου να υλοποιήσει κατάλογο, ώστε να μειωθεί η κίνηση του πρωτοκόλλου συνοχής, ειδικά σε συστήματα με 4–8 επεξεργαστές.

Στο άλλο άκρο, έχουμε επεξεργαστές με πολλούς έως πάρα πολλούς πυρήνες οι οποίοι περιλαμβάνουν και το τμήμα του δικτύου διασύνδεσης. Ένα χαρακτηριστικό παράδειγμα είναι ο επεξεργαστής Epiphany της εταιρείας Adapteva. Ο επεξεργαστής αυτός αποτελεί τη βάση για ένα από τα δημοφιλέστερα παράλληλα συστήματα μικρού μεγέθους, μικρής κατανάλωσης ενέργειας και ανοιχτής αρχιτεκτονικής. Πρόκειται για το σύστημα Parallella, το οποίο καταλαμβάνει επιφάνεια όσο μία πιστωτική κάρτα και διαθέτει δύο επεξεργαστές: ένα διπύρηνο επεξεργαστή τύπου ARM V9 και τον Epiphany ως συνεπεξεργαστή. Στη βασική του έκδοση (III) ο επεξεργαστής Epiphany διαθέτει 16 πυρήνες, ενώ έχει ήδη κυκλοφορήσει σε περιορισμένες ποσότητες ο Epiphany-IV με 64 πυρήνες. Κάθε πυρήνας είναι ένας μικρός επεξεργαστής RISC, με χρονισμό μέχρι 1GHz και κατανάλωση μόλις 25mW.

Οι πυρήνες του επεξεργαστή Epiphany είναι οργανωμένοι σε ένα διδιάστατο πλέγμα το οποίο βρίσκεται μέσα στο ολοκληρωμένο κύκλωμα (network-on-chip, NOC). Ο Epiphany III φαίνεται στο Σχ. 3.21 όπου απεικονίζεται το πλέγμα 4×4 και η χονδρική δομή κάθε κόμβου. Ο επεξεργαστής διαθέτει επιπλέον 4 θύρες επικοινωνίας, στις οποίες ενσωματώνονται κανάλια προς το βορρά, την ανατολή, τη δύση και το νότο (NEWS) προκειμένου να συνδέσει κανείς επιπλέον επεξεργαστές μεταξύ τους για τη δημιουργία μεγαλύτερων πλεγμάτων (μέχρι $64 \times 64 = 4096$ πυρήνες μπορεί να υπάρχουν σε ένα τέτοιο σύστημα). Το δίκτυο αποτελείται από τρία αυτόνομα υποδίκτυα με διαφορετικές ταχύτητες: τα cMesh και xMesh χρησιμοποιούνται για απομακρυσμένες εγγραφές εντός και εκτός του επεξεργαστή, αντίστοιχα, ενώ το rMesh και αναγνώσεις.



Σχήμα 3.21 *Η οργάνωση του 16-πύρηνου επεξεργαστή Epiphany III.*

Κάθε πυρήνας διαθέτει 32KiB τοπική μνήμη και καθόλου κρυφή μνήμη. Σε όλο το δίκτυο υποστηρίζεται κοινόχρηστος χώρος διευθύνσεων, αλλά ο παράγοντας NUMA δεν είναι ιδιαίτερα μικρός. Ενώ η τοπική μνήμη γράφεται σε 1 κύκλο ρολογιού, η εγγραφή απομακρυσμένου δεδομένου (εντός του επεξεργαστή) απαιτεί $1,5 \times D$ κύκλους, όπου D είναι η απόσταση που πρέπει να διανυθεί στο δίκτυο. Έτσι, για παράδειγμα, χρειάζονται 10,5 κύκλοι για να γίνει διαγώνια εγγραφή από τον πυρήνα επάνω αριστερά στην μνήμη του κάτω δεξιά.

3.9 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις

Στο κεφάλαιο αυτό μελετήσαμε τους πολυεπεξεργαστές κατανεμημένης μνήμης, μία αρχιτεκτονική στην οποία η μνήμη έχει τμηματοποιηθεί και διαμοιραστεί στους επεξεργαστές. Η οργάνωση ολοκληρώνεται με ένα δίκτυο διασύνδεσης μεταξύ των επεξεργαστών, μέσω του οποίου μπορούν να ανταλλαχθούν μηνύματα και να μεταφερθούν δεδομένα.

Στην κλασική τους μορφή, υπάρχει μία ιδιωτική μνήμη ανά επεξεργαστή. Στους ομαδοποιημένους πολυεπεξεργαστές, όμως, υπάρχει μία κοινή μνήμη ανά ομάδα επεξεργαστών (κάθε ομάδα είναι, δηλαδή, ένας συμμετρικός πολυεπεξεργαστής) και το δίκτυο διασύνδεσης συνδέει, πλέον, τις ομάδες. Ανεξάρτητα από την τελική τους μορφή, τα συστήματα αυτά μπορούν με την προσθήκη κατάλληλων μονάδων να υποστηρίξουν διαφανώς έναν κοινό χώρο διευθύνσεων μεταξύ όλων των επεξεργαστών. Μπορεί, δηλαδή, να γίνει εξομοίωση κοινής μνήμης σε επίπεδο υλικού, κάτι που είναι γνωστό ως κατανεμημένη κοινή μνήμη. Μία τέτοια οργάνωση εμπλέκει κρυφές μνήμες για αποθήκευση απομακρυσμένων

δεδομένων και, κατά συνέπεια, χρήση πρωτοκόλλων συνοχής. Μία αναλυτική επισκόπηση των συστημάτων και των προβλημάτων της κατανεμημένης κοινής μνήμης, τόσο σε επίπεδο υλικού όσο και λογισμικού, δίνεται στο [KKLR00].

Σε όλα αυτά τα συστήματα, το σημαντικότερο τμήμα είναι το δίκτυο διασύνδεσης, αφού σε αυτό στηρίζεται η επικοινωνία των επεξεργαστών και άρα, η «ενότητα» του συστήματος. Το δίκτυο θα πρέπει να πληροί κάποιες επιθυμητές προϋποθέσεις, τόσο για λόγους κόστους υλοποίησης όσο και για λόγους ταχύτητας επικοινωνίας. Οι προϋποθέσεις περιλαμβάνουν τον χαμηλό βαθμό των κορυφών και τη χαμηλή διάμετρο. Τα δύο αυτά ζητούμενα είναι, όμως, τις περισσότερες φορές αλληλοσυγκρουόμενα και στα πρακτικά δίκτυα υπάρχει κάποιος συμβιβασμός. Είχαμε την ευκαιρία να δούμε αρκετά δίκτυα διασύνδεσης, όπως τους πλήρεις και τους γραμμικούς γράφους, τους δακτυλίους, καθώς επίσης και τα πολυδιάστατα δίκτυα του πλέγματος και του torus, τα οποία μαζί με τον υπερκύβο είναι ιδιαίτερα δημοφιλή στην πράξη. Για τον αναγνώστη ο οποίος ενδιαφέρεται να εμβαθύνει στην τοπολογία των δικτύων, προτείνεται το βιβλίο του J. Xu [Xu10]. Μολονότι παλαιότερο, το βιβλίο του F.T. Leighton [Leig92] ασχολείται επίσης κατά μεγάλο μέρος, με τοπολογία και προχωρημένες τοπολογικές και επικοινωνιακές ιδιότητες σημαντικών δικτύων διασύνδεσης.

Μεγάλο ρόλο στη συμπεριφορά και τις επιδόσεις ενός δικτύου διασύνδεσης παίζει η μεταγωγή που χρησιμοποιείται. Η δημοφιλέστερη μεταγωγή στους αρχικούς πολυεπεξεργαστές ήταν η μεταγωγή πακέτου ή μηνύματος. Πιο πρόσφατα, όμως, έχουν επικρατήσει η μεταγωγή wormhole και virtual cut-through. Με τις δύο αυτές μεταγωγές είναι χαρακτηριστικό ότι ο χρόνος επικοινωνίας γίνεται πολλές φορές σχεδόν ανεξάρτητος από την απόσταση που διανύει το μήνυμα (αρκεί να μην υπάρχει μεγάλη κίνηση στο δίκτυο). Έτσι, γίνεται λιγότερο σημαντική η αναζήτηση δικτύων με χαμηλή διάμετρο. Αυτά τα είδη μεταγωγής απαιτούν εξειδικευμένους διαδρομητές με ιδιαίτερες αρχιτεκτονικές, ώστε να μπορούν να υποστηρίξουν επικοινωνίες πολύ χαμηλών καθυστερήσεων αλλά και προχωρημένες τεχνικές, όπως αυτή των εικονικών καναλιών.

Η αρχιτεκτονική του διαδρομητή αποτελεί το κεντρικό θέμα του βιβλίου των Dally και Towles [DaT004]. Το βιβλίο που προσεγγίζει τα περισσότερα από τα θέματα που μελετήσαμε εδώ, τόσο για τα δίκτυα διασύνδεσης αυτού του κεφαλαίου όσο και για πολλά από τα δίκτυα που είδαμε στο προηγούμενο κεφαλαίο για συστήματα κοινόχρηστης μνήμης, είναι το βιβλίο των Duato, Yalamanchili και Ni [DYN03].

Το βιβλίο των Culler, Singh και Gupta [CSG99] αποτελεί ένα από τα σημαντικότερα βιβλία που έχουν γραφτεί για το γενικότερο θέμα της αρχιτεκτονικής των παράλληλων συστημάτων. Καλύπτει σε μεγάλο βάθος τόσο συστήματα κατανεμημένης όσο και κοινόχρηστης μνήμης, με εξαίρεση τα πολυπύρηνα συστήματα που αποτελούν πιο πρόσφατη εξέλιξη.

Για τα πιο πρόσφατα συστήματα που συναντήσαμε στο κεφάλαιο και τα οποία δεν καλύπτονται στην παραπάνω βιβλιογραφία, υπάρχουν αρκετές πηγές πληροφόρησης,

πολλές εκ των οποίων είναι διαθέσιμες και στον παγκόσμιο ιστό. Τα άρθρα [CKDL10, CoHu07] περιγράφουν αναλυτικά την οργάνωση των επεξεργαστών Opteron. Μία εποπτική ματιά στους επεξεργαστές Xeon αλλά κυρίως στη λειτουργία του δικτύου QuickPath που χρησιμοποιούν (το οποίο είναι αντίστοιχο του HyperTransport των Opteron), παρέχεται στο white paper της εταιρείας Intel [Inte09]. Τέλος, περισσότερες πληροφορίες για τον επεξεργαστή Epiphany αλλά και το μίνι-παράλληλο σύστημα Parallella μπορείτε να βρείτε από την εταιρεία Adapteva [Adap14b, Adap14a].

Προβλήματα

3.1 – Ένα χρήσιμο μέγεθος σε ένα δίκτυο είναι οι *μέσες αποστάσεις* των κορυφών. Η μέση απόσταση, $AD(v)$, για έναν κόμβο v, ορίζεται ως ο μέσος όρος των αποστάσεων από τις υπόλοιπες κορυφές, δηλαδή, $AD(v) = \sum_u d(v,u)/(N-1)$, όπου N είναι ο αριθμός των κορυφών και $d(v,u)$ η απόσταση μεταξύ v και u. Μπορείτε να βρείτε τη μέση απόσταση για τον κόμβο 0 στους γράφους που περιέχει ο Πίνακας 3.1;

3.2 – Σχεδιάστε τον 4-διάστατο κύβο Q_4 με πλήρη ονόματα κόμβων, και στη συνέχεια δώστε τέσσερις διαφορετικές διαδρομές μεταξύ των κόμβων 2 και 10 (κάνοντας διόρθωση των bits στα οποία διαφέρουν). Ποια από αυτές θα ακολουθούσε η διαδρόμηση e-cube;

3.3 – Πώς θα ήταν οι διαδρομές στον υπερκύβο του Σχ. 3.16, αν ακολουθούσαμε αυστηρά τη διαδρόμηση e-cube;

3.4 – Έστω $a = (a_{d-1}a_{d-2}\cdots a_0)_2$ ένας κόμβος σε έναν d-διάστατο υπερκύβο. Ποιος κόμβος απέχει τη μέγιστη δυνατή απόσταση από αυτόν;

3.5 – Πόσες διαφορετικές διαδρομές *ελάχιστου* μήκους υπάρχουν μεταξύ δύο δεδομένων κόμβων σε έναν υπερκύβο;

3.6 – Αντιστοιχίστε έναν δακτύλιο 12 κόμβων σε έναν 4-κύβο και ένα πλέγμα $2 \times 2 \times 6$ σε έναν 5-κύβο.

4 Προγραμματισμός με Κοινόχρηστο Χώρο Διευθύνσεων

Με το κεφάλαιο αυτό, αλλάζουμε θεματολογία και μπαίνουμε στον χώρο του προγραμματισμού των παράλληλων συστημάτων. Σε αντίθεση με τους κλασικούς σειριακούς υπολογιστές, ο προγραμματιστής είναι απαραίτητο να έχει πιο άμεση γνώση της αρχιτεκτονικής του συστήματος προκειμένου τα προγράμματά του να πετυχαίνουν τις επιθυμητές επιδόσεις. Θα ξεκινήσουμε με κάποιες γενικές έννοιες και χαρακτηριστικά του παράλληλου προγραμματισμού και θα δούμε πώς μπορεί κάποιος να εκμεταλλευτεί την ύπαρξη πολλαπλών επεξεργαστών ή πυρήνων από την πλευρά του λογισμικού.

Το κεφάλαιο αυτό, όμως, εστιάζεται στο πρώτο μοντέλο παράλληλου προγραμματισμού, το μοντέλο κοινόχρηστου χώρου διευθύνσεων. Το μοντέλο αυτό είναι επίσης γνωστό ως μοντέλο κοινόχρηστων μεταβλητών (shared variables model), και, αν και καταχρηστικά, ως μοντέλο κοινόχρηστης μνήμης. Ο σκοπός μας θα είναι να δούμε τις συντακτικές δομές και τις ευκολίες που παρέχονται συνήθως για τον προγραμματισμό σε κοινό χώρο διευθύνσεων. Επίσης, θα μάθουμε βασικές προγραμματιστικές τεχνικές για το μοντέλο αυτό, μέσα από το σχεδιασμό απλών και πρακτικών προγραμμάτων.

Η διαδρομή μας θα στηριχτεί επάνω στις δύο πιο σημαντικές και ευρέως χρησιμοποιούμενες προσεγγίσεις του μοντέλου αυτού: το προγραμματισμό με τα νήματα posix και με το OpenMP.

Το μοντέλο *κοινόχρηστου χώρου διευθύνσεων* (shared address space model) είναι το πρώτο από τα δύο μοντέλα παράλληλου προγραμματισμού που θα μελετήσουμε, τα οποία προϋποθέτουν την ύπαρξη πολλαπλών οντοτήτων εκτέλεσης (ροών ελέγχου) που εργάζονται ταυτόχρονα. Ως οντότητες εκτέλεσης θεωρούμε τις διεργασίες ή τα νήματα. Στη μελέτη μας στηριζόμαστε στη βιβλιοθήκη των νημάτων POSIX, και θεωρούμε ότι οι οντότητες εκτέλεσης είναι κατά κύριο λόγο νήματα, κάτι που ταιριάζει απόλυτα με την οργάνωση των σύγχρονων πολυπύρηνων επεξεργαστών.

Στο μοντέλο κοινόχρηστου χώρου διευθύνσεων, όπως και στο μοντέλο μεταβίβασης μηνυμάτων που θα δούμε αργότερα, κάθε επεξεργαστής αναλαμβάνει την εκτέλεση μίας διαφορετικής οντότητας. Από τη στιγμή που οι επεξεργαστές δεν λειτουργούν συγχρονισμένα, δηλαδή δεν εκτελούν την ίδια εντολή, την ίδια χρονική στιγμή, εμφανίζεται και η έννοια της επικοινωνίας / συνεργασίας των οντοτήτων, η οποία στο μοντέλο κοινόχρηστου χώρου διευθύνσεων γίνεται μέσω κοινόχρηστων ή διαμοιραζόμενων μεταβλητών. Αυτά, αλλά και άλλα στοιχεία που κάνουν την εμφάνισή τους κατά τον προγραμματισμό στο μοντέλο κοινόχρηστου χώρου διευθύνσεων θα τα δούμε για πρώτη φορά στην Ενότητα 4.2.

Τις προγραμματιστικές δομές που απαιτούνται για τη δημιουργία οντοτήτων και για τον ορισμό μεταβλητών κοινών μεταξύ τους, θα τις δούμε με μεγαλύτερη λεπτομέρεια στην Ενότητα 4.3. Στη συνέχεια, οι Ενότητες 4.4 και 4.5 ασχολούνται με την επίλυση κάποιων προβλημάτων που δημιουργούνται λόγω της ύπαρξης πολλαπλών ροών εκτέλεσης και κοινόχρηστων μεταβλητών. Τα προβλήματα αυτά είναι ο αμοιβαίος αποκλεισμός και ο συγχρονισμός.

Μετά τη γενική εισαγωγή στο μοντέλο του κοινόχρηστου χώρου διευθύνσεων (Ενότητες 4.2–4.5), οι Ενότητες 4.6–4.8 θα μας μυήσουν στον τρόπο σκέψης του συγκεκριμένου μοντέλου, κάνοντας χρήση των νημάτων POSIX. Η ενότητες αυτές διδάσκουν βασικές τεχνικές μέσα από μικρές εφαρμογές. Συγκεκριμένα, θα δούμε την παραλληλοποίηση απλών ή πολλαπλών βρόχων `for`, με διάφορους τρόπους. Επίσης, θα γνωρίσουμε την τεχνική της αυτοδρομολόγησης η οποία είναι χρήσιμη σε περιπτώσεις όπου απαιτείται μία αυτόματη ισοκατανομή της εργασίας.

Στην Ενότητα 4.9 θα γνωρίσουμε τον πλέον δημοφιλή, στις μέρες μας, τρόπο προγραμματισμού των μηχανών κοινόχρηστης μνήμης, το OpenMP. Αν και στηρίζεται, έστω και έμμεσα, στα νήματα, πρόκειται για ένα εντελώς διαφορετικό και πολύ πιο προσιτό στυλ παράλληλου προγραμματισμού. Συγκεκριμένα, ο προγραμματιστής προσδιορίζει (μέσω οδηγιών) ποιο τμήμα του κώδικα πρέπει να παραλληλοποιηθεί, αλλά την παραλληλοποίηση την αναλαμβάνει ο μεταφραστής. Η επιτυχία του OpenMP συνδυάστηκε με την καθιέρωση των πολυπύρηνων επεξεργαστών.

Τέλος, στην Ενότητα 4.10 θα κλείσουμε με μια ματιά σε κάποιες ιδιαίτερες εφαρμογές, οι οποίες δεν παραλληλοποιούνται εύκολα. Για τη μελέτη τους θα χρησιμοποιήσουμε το OpenMP διότι παρέχει έναν μηχανισμό που πολλές φορές είναι πολύ αποτελεσματικός για τέτοιου είδους προβλήματα.

4.1 Γενικά περί παράλληλων προγραμμάτων

Σε κάποιον που δεν έχει προγραμματίσει ποτέ παράλληλα, η ενστικτώδης απορία είναι, με ποιο τρόπο θα μπορέσουν να εμπλακούν πολλαπλοί επεξεργαστές ή πυρήνες[1] στο ένα και μοναδικό πρόγραμμα που θα δημιουργήσει. Για να το δούμε αυτό, θα θυμηθούμε κάποια βασικά πράγματα που έχουν σχέση με την εκτέλεση των προγραμμάτων.

Ως *πρόγραμμα* θεωρούμε τον κώδικα (εντολές) που έχει γραφτεί προκειμένου να φέρει εις πέρας έναν υπολογισμό. Συνήθως, η αρχική μορφή ενός προγράμματος δίνεται σε μία γλώσσα προγραμματισμού υψηλού επιπέδου, και στη συνέχεια μετατρέπεται από μεταφραστή σε ισοδύναμο πρόγραμμα σε γλώσσα μηχανής, γνωστό και ως *εκτελέσιμο* (executable). Τα προγράμματα βρίσκονται αποθηκευμένα σε κάποια συσκευή αποθήκευσης και προκειμένου να εκτελεστούν πρώτα *φορτώνονται* από το λειτουργικό σύστημα στην κύρια μνήμη του υπολογιστή, και στην συνέχεια γίνεται η έναρξη της εκτέλεσης. Ένα πρόγραμμα που έχει φορτωθεί και έχει ξεκινήσει την εκτέλεσή του ονομάζεται *διεργασία* (process). Ενώ το πρόγραμμα το δημιουργεί ο προγραμματιστής, τις διεργασίες τις διαχειρίζεται το λειτουργικό σύστημα και αναθέτει την εκτέλεσή τους στους επεξεργαστές.

Μία διεργασία, όμως, αποτελείται από περισσότερα μέρη από ότι το αρχικό πρόγραμμα. Κατ' ελάχιστον, κάθε διεργασία διαθέτει:

- Ένα τμήμα κώδικα (οι εντολές του προγράμματος),
- ένα τμήμα δεδομένων (πρόκειται για τις *καθολικές* μεταβλητές του προγράμματος),
- μία στοίβα (stack) η οποία χρησιμοποιείται για την αποθήκευση *τοπικών* μεταβλητών κατά την κλήση συναρτήσεων,
- έναν μετρητή προγράμματος (program counter), δηλαδή έναν καταχωρητή που προσδιορίζει ποια είναι η επόμενη εντολή που θα εκτελεστεί.

Τα δύο πρώτα βρίσκονται έτοιμα μέσα στο αποθηκευμένο πρόγραμμα και κατά τη φόρτωση του προγράμματος απλά αντιγράφονται σε κάποια σημεία στην κύρια μνήμη. Τα άλλα δύο δημιουργούνται από το λειτουργικό σύστημα τη στιγμή της φόρτωσης, και ο ρόλος τους είναι να υποστηρίξουν την εκτέλεση. Ο μετρητής επιλέγει την επόμενη εντολή, και η στοίβα παρέχει χώρο για αποθήκευση προσωρινών δεδομένων (τοπικών μεταβλητών). Ο συνδυασμός μετρητή προγράμματος και στοίβας ονομάζεται *νήμα* (thread). Όταν λέμε ότι εκτελείται η διεργασία, ουσιαστικά το νήμα είναι αυτό που λειτουργεί, χρησιμοποιώντας τις εντολές και τις καθολικές μεταβλητές της. Το νήμα αυτό ονομάζεται και *κύριο* (main) ή *αρχικό* (initial) νήμα της διεργασίας.

[1]Χάριν ευκολίας, στο κεφάλαιο αυτό θα θεωρούμε τους όρους επεξεργαστής και πυρήνας ως ισοδύναμους, εφόσον δεν υπάρχει περίπτωση σύγχυσης.

Κατά τη διάρκεια της εκτέλεσής της, μία διεργασία μπορεί να δημιουργήσει νέες διεργασίες ή/και νέα νήματα:

- Η κλασική κλήση `fork()` των συστημάτων τύπου Unix, δημιουργεί ένα πλήρες αντίγραφο (κώδικας, δεδομένα, στοίβα κλπ) της τρέχουσας διεργασίας. Παρ' όλα αυτά, πρόκειται για εντελώς ανεξάρτητη διεργασία που δεν μοιράζεται τίποτε με την αρχική.
- Το αρχικό νήμα μπορεί να δημιουργήσει νέα νήματα, με διάφορους τρόπους. Προσέξτε, όμως, ότι ανεξάρτητα με το πώς θα δημιουργηθούν τα επιπλέον νήματα, θα εκτελούνται με βάση το ίδιο τμήμα κώδικα και ενεργώντας πάνω στο ίδιο τμήμα καθολικών δεδομένων. Οι στοίβες τους θα είναι διαφορετικές, βέβαια.

Στα παραπάνω βρίσκεται και η απάντηση στο ερώτημα πώς εκμεταλλευόμαστε προγραμματιστικά ένα παράλληλο σύστημα. Προκειμένου ένα πρόγραμμα να στρατολογήσει πολλαπλούς επεξεργαστές ή πυρήνες για την εκτέλεσή του, θα πρέπει να δημιουργήσει πολλαπλές *οντότητες εκτέλεσης* (execution entitites), δηλαδή πολλαπλές διεργασίες ή πολλαπλά νήματα. Το λειτουργικό σύστημα, στη συνέχεια, θα αναθέσει σε διαφορετικούς πυρήνες να μοιραστούν τις οντότητες και να τις εκτελέσουν ταυτόχρονα.

Η επιλογή του τύπου των οντοτήτων εκτέλεσης είναι σημαντική. Η δημιουργία και η διαχείριση διεργασιών είναι γενικά πολύ πιο χρονοβόρα από τη δημιουργία νημάτων. Είναι χαρακτηριστικό ότι τα νήματα λέγονται μερικές φορές και «ελαφριές» διεργασίες, και γενικότερα προτιμούνται στον παράλληλο προγραμματισμό επιδόσεων. Οι διεργασίες δεν έχουν τίποτε κοινό μεταξύ τους, σε αντίθεση με τα νήματα τα οποία μοιράζονται τον ίδιο κώδικα και τις ίδιες καθολικές μεταβλητές. Αυτό σημαίνει ότι όποιος προγραμματίζει με διεργασίες, θα πρέπει να χρησιμοποιήσει επιπρόσθετους μηχανισμούς για να επικοινωνούν αυτές μεταξύ τους. Από την άλλη, υπάρχει πολύ μεγαλύτερη απομόνωση ανάμεσα στις διεργασίες απ' ότι στα νήματα, κάτι που παρέχει μεγαλύτερη ασφάλεια αλλά και ανοχή σε σφάλματα. Τέλος, τα νήματα δεν είναι εντελώς αυτόνομες οντότητες, καθώς υπάρχουν και ανήκουν σε μία διεργασία. Έτσι, σε μερικές περιπτώσεις δεν μπορούν εύκολα να χρησιμοποιηθούν νήματα, όπως για παράδειγμα στον προγραμματισμό συστάδων όπου εμπλέκονται ανεξάρτητοι υπολογιστές.

4.2 Βασικές δομές του μοντέλου

Όπως κάθε παράλληλο πρόγραμμα, έτσι και ένα πρόγραμμα στο μοντέλο κοινόχρηστου χώρου διευθύνσεων αποτελείται από μία συλλογή από οντότητες εκτέλεσης. Κάθε σύστημα, λοιπόν, που προσφέρει τη δυνατότητα προγραμματισμού με κοινόχρηστο χώρο διευθύνσεων (αλλά και με μεταβίβαση μηνυμάτων, που θα δούμε στο επόμενο κεφάλαιο), πρέπει να

παρέχει δομές για τη *δημιουργία* και τη *διαχείριση* των οντοτήτων εκτέλεσης. Οι γενικές έννοιες που θα συναντήσουμε στο κεφάλαιο αυτό ισχύουν και για τα δύο είδη οντοτήτων εκτέλεσης. Για πρακτικούς λόγους θα επικεντρωθούμε σε ένα είδος οντότητας, στα νήματα, που είναι και η πολύ πιο συνηθισμένη περίπτωση στην πράξη.

Μία συνήθης τακτική για ένα παράλληλο πρόγραμμα είναι να ξεκινά την εκτέλεσή του κάνοντας κάποιες αρχικοποιήσεις, και στη συνέχεια να δημιουργεί ένα πλήθος άλλων νημάτων προκειμένου να εκτελέσουν παράλληλα ένα χρονοβόρο τμήμα του κώδικα. Η εργασία μοιράζεται στα νήματα αυτά, και μόλις ολοκληρωθεί, τερματίζονται τα νήματα που δημιουργήθηκαν. Ανάλογα με την εφαρμογή, το σενάριο πιθανώς να χρειαστεί να επαναληφθεί αρκετές φορές, εφόσον υπάρχουν πολλά διαφορετικά τμήματα του κώδικα που χρήζουν παράλληλης εκτέλεσης. Αυτή είναι η λεγόμενη *στρατηγική fork-join*, όπου τα παράλληλα νήματα δημιουργούνται (fork) όποτε χρειάζεται και καταστρέφονται (join) με το τέλος του εκάστοτε υπολογισμού, ενώ τα ενδιάμεσα τμήματα του κώδικα εκτελούνται σειριακά από το αρχικό νήμα.

Από τη στιγμή που θα δημιουργηθούν τα νήματα, το ερώτημα είναι πώς θα επικοινωνήσουν προκειμένου να συνεργαστούν για την ολοκλήρωση του υπολογισμού. Στο μοντέλο κοινόχρηστου χώρου διευθύνσεων αυτό γίνεται μέσω *κοινόχρηστων ή διαμοιραζόμενων* μεταβλητών (shared variables), δηλαδή μεταβλητών οι οποίες μπορούν να προσπελαστούν και να τροποποιηθούν από όλα τα νήματα, ακόμα και αν αυτά εκτελούνται σε διαφορετικούς επεξεργαστές. Επομένως, στο μοντέλο μας θα πρέπει να υπάρχουν δομές οι οποίες δημιουργούν τις κοινόχρηστες μεταβλητές και τις διαχωρίζουν από τις *ιδιωτικές* μεταβλητές (private variables) του κάθε νήματος. Η ιδέα είναι εντελώς ανάλογη με τις *τοπικές* (local) και τις *καθολικές* (global) μεταβλητές σε ένα απλό σειριακό πρόγραμμα. Η κάθε συνάρτηση/διαδικασία στο σειριακό πρόγραμμα διαθέτει τοπικές μεταβλητές σε αναλογία με τις ιδιωτικές μεταβλητές που διαθέτει κάθε νήμα στο παράλληλο πρόγραμμα. Οι καθολικές μεταβλητές που τις «βλέπουν» όλες οι συναρτήσεις είναι ανάλογες με τις κοινόχρηστες μεταβλητές που τις «βλέπουν» όλα τα νήματα. Η ουσιαστική διαφορά είναι, βέβαια, ότι τα νήματα εκτελούνται παράλληλα.

Κάθε νήμα μπορεί να εκτελείται σε διαφορετικό πυρήνα (ή επεξεργαστή) και επειδή οι πυρήνες είναι ανεξάρτητοι μεταξύ τους, και καθένας εκτελεί εντολές με τον δικό του ρυθμό, τα νήματα εκτελούνται *ασύγχρονα*. Ο όρος «ασύγχρονα» εμπεριέχει και κάποια τυχαιότητα, υπό την έννοια ότι δεν είναι προβλέψιμη η χρονική συμπεριφορά του κάθε νήματος. Το πότε θα εκτελεστεί η επόμενη εντολή σε ένα νήμα εξαρτάται από το συνολικό φόρτο του πυρήνα που το έχει αναλάβει. Η λεπτομέρεια αυτή είναι υπεύθυνη για σημαντικά προβλήματα που εμφανίζονται κατά την εκτέλεση του παράλληλου προγράμματος.

Ας πάρουμε ένα απλό παράδειγμα, τον υπολογισμό του μέσου όρου (`avg`) των 10 στοιχείων ενός διανύσματος A. Έστω ότι τον υπολογισμό θα τον αναλάβουν δύο νήματα, T_1 και T_2. Η λογική τακτική είναι να αναλάβει καθένα να υπολογίσει τη συνεισφορά 5 στοιχείων του διανύσματος στον μέσο όρο, όπως στο Πρόγρ. 4.1. Οι μεταβλητές `A` και `avg`

Νήμα T_1

```
1 /* SHARED:  A[10], avg
2  * PRIVATE: i, q1
3  */
4 q1 = 0;
5 for (i = 0; i < 5; i++)
6   q1 += A[i];
7 q1 = q1 / 10.0;
8 avg = avg + q1;
```

Νήμα T_2

```
1 /* SHARED:  A[10], avg
2  * PRIVATE: i, q2
3  */
4 q2 = 0;
5 for (i = 5; i < 10; i++)
6   q2 += A[i];
7 q2 = q2 / 10.0;
8 avg = avg + q2;
```

Πρόγραμμα 4.1 *Δύο νήματα που συνεργάζονται για τον υπολογισμό του μέσου όρου ενός διανύσματος A, 10 στοιχείων.*

θα πρέπει να έχουν δηλωθεί με κάποιον τρόπο ως κοινόχρηστες, ώστε να μπορούν να είναι προσπελάσιμες και από τα δύο νήματα. Επίσης, η avg θα πρέπει να έχει αρχικοποιηθεί στην τιμή 0. Οι υπόλοιπες μεταβλητές είναι ιδιωτικές στο κάθε νήμα.

Ενώ ο παραπάνω κώδικας φαίνεται απόλυτα λογικός και σωστός, εντούτοις δεν δίνει πάντα το σωστό αποτέλεσμα! Ο λόγος είναι η ασύγχρονη εκτέλεση των νημάτων. Κάθε νήμα εκτελεί τις εντολές που έχουν καθοριστεί στον κώδικα με τη σειρά. Όμως, δεν μπορεί να προβλεφθεί η χρονική στιγμή εκτέλεσης της κάθε εντολής. Είναι δυνατόν το νήμα T_1 να έχει ολοκληρώσει την εκτέλεσή του και το νήμα T_2 να μην έχει ακόμη ξεκινήσει τη δική του εκτέλεση λόγω, π.χ. αυξημένου φόρτου στον πυρήνα που το έχει αναλάβει. Η ασύγχρονη αυτή εκτέλεση των διεργασιών δημιουργεί σοβαρά προβλήματα, που δεν είναι ορατά με την πρώτη ματιά.

Το πρώτο πρόβλημα σχετίζεται με την προσπέλαση των κοινόχρηστων μεταβλητών. Η γραμμή 8 του κώδικα των δύο νημάτων που τροποποιεί την κοινόχρηστη μεταβλητή avg στην πράξη δεν εκτελείται στιγμιαία αλλά μέσα από μία σειρά βημάτων τα οποία υλοποιούνται στη γλώσσα μηχανής του επεξεργαστή:

8α Μεταφορά της τιμή της avg σε έναν καταχωρητή.

8β Μεταφορά της τιμής της q1 (αντίστοιχα, της q2, για τον πυρήνα που εκτελεί το T_2) σε έναν άλλο καταχωρητή.

8γ Πρόσθεση των δύο καταχωρητών.

8δ Αποθήκευση του αποτελέσματος στη θέση μνήμης που φυλάσσεται η avg.

Υποθέστε ότι τη χρονική στιγμή t τα δύο νήματα έχουν ολοκληρώσει την εκτέλεση των γραμμών 1–7 του κώδικά τους, όπου υπολόγισαν q1 = 30 και q2 = 70 και αρχίζουν να εκτελούν τη γραμμή 8. Το αποτέλεσμα θα πρέπει κανονικά να βγει ίσο με 100. Όμως, η ασύγχρονη εκτέλεση των νημάτων μπορεί να οδηγήσει σε ανεπιθύμητα αποτελέσματα. Μερικά από τα δυνατά σενάρια δίνονται στον Πίνακα 4.1.

Πίνακας 4.1 *Πιθανά σενάρια κατά την εκτέλεση της 8ης εντολής των νημάτων στο Πρόγρ. 4.1*

Χρονική	1ο ΣΕΝΑΡΙΟ		2ο ΣΕΝΑΡΙΟ		3ο ΣΕΝΑΡΙΟ	
στιγμή	T_1	T_2	T_1	T_2	T_1	T_2
t	8α	-	8α	8α	-	8α
$t+1$	8β	-	8β	8β	-	8β
$t+2$	-	-	8γ	8γ	-	8γ
$t+3$	-	8α	-	-	-	8δ
$t+4$	-	8β	-	-	8α	-
$t+5$	-	8γ	8δ	-	8β	-
$t+6$	8γ	8δ	-	-	8γ	-
$t+7$	8δ	-	-	8δ	8δ	-
Αποτέλεσμα	**30**		**70**		**100**	

Στο πρώτο σενάριο, το νήμα T_1 διαβάζει την τιμή της avg, η οποία έχει τιμή 0, και της q1. Καθυστερώντας, όμως, να εκτελέσει την πρόσθεση (με αποτέλεσμα 30), και την αποθήκευση του αποτελέσματος στη μνήμη, το νήμα T_2 προλαβαίνει να διαβάσει την τιμή της avg (0) και να ολοκληρώσει πλήρως τη δική του εντολή, αποθηκεύοντας στη μνήμη την τιμή 70, τη χρονική στιγμή $t+6$. Την ίδια χρονική στιγμή το T_1 εκτελεί την πρόσθεση και αμέσως μετά αποθηκεύει το δικό του αποτέλεσμα, και η τελική τιμή της avg καταλήγει να είναι το 30, το οποίο προφανώς είναι λάθος.

Το δεύτερο σενάριο κάνει ακριβώς το αντίθετο. Πριν προλάβει το νήμα T_2 να αποθηκεύσει το αποτέλεσμά του, το νήμα T_1 έχει αποθηκεύσει το δικό του τη στιγμή $t+5$. Στη συνέχεια το T_1 εκτελεί την αποθήκευση και το τελικό αποτέλεσμα γίνεται ίσο με 70, πάλι λανθασμένα. Στο τρίτο σενάριο, το νήμα T_2 ολοκληρώνει τη λειτουργία του πριν το T_1 διαβάσει την τιμή της μεταβλητής avg. Έτσι, το T_1 διαβάζει τη σωστή τιμή της μεταβλητής, προσθέτει το q1 και αποθηκεύει τη σωστή τελική τιμή στη μνήμη.

Το πρόβλημα εμφανίζεται κατά την προσπέλαση των κοινόχρηστων μεταβλητών, εξαιτίας του απρόβλεπτου χρονισμού της εκτέλεσης των νημάτων. Το τρίτο σενάριο δούλεψε διότι το ένα νήμα πρόλαβε να ολοκληρώσει την τροποποίηση της avg, πριν ξεκινήσει να την τροποποιεί κάποιο άλλο. Τα προβλήματα θα μπορούσαν εύκολα να αποφευχθούν εάν *δεν επιτρεπόταν στον κώδικα η ταυτόχρονη τροποποίηση της κοινής μεταβλητής από παραπάνω από ένα νήμα*. Ο περιορισμός αυτός είναι ο γνωστός *αμοιβαίος αποκλεισμός* (mutual exclusion). Κάθε γλώσσα ή σύστημα προγραμματισμού για το μοντέλο του κοινόχρηστου χώρου διευθύνσεων παρέχει, επομένως, δομές και εντολές ή συναρτήσεις που μπορούν να χρησιμοποιηθούν για την επίτευξη αμοιβαίου αποκλεισμού. Οι δομές αυτές, όπως θα δούμε, χρησιμοποιούνται κάθε φορά σε τμήματα του κώδικα όπου απαιτείται η

προσπέλαση κάποιας κοινόχρηστης μεταβλητής.

Η μη συγχρονισμένη εκτέλεση των διεργασιών ευθύνεται και για ένα δεύτερο πρόβλημα. Ας υποθέσουμε ότι στη συνέχεια του κώδικα στο Πρόγρ. 4.1, το νήμα T_1 χρειάζεται το τετράγωνο του μέσου όρου για κάποιους άλλους υπολογισμούς, οπότε στον κώδικά του υπάρχει και μία 9η γραμμή:

```
9  x = avg*avg;
```

Μπορεί να μην είναι προφανές, αλλά όπως είναι δοσμένος ο κώδικας, ακόμα και με τη χρήση αμοιβαίου αποκλεισμού δεν μπορούμε να είμαστε σίγουροι ότι η x θα λάβει τη σωστή τιμή! Αυτό συμβαίνει διότι, όταν το νήμα T_1 αρχίζει την εκτέλεση της 9ης γραμμής, το νήμα T_2 μπορεί να μην έχει καν αρχίσει την εκτέλεση της δικής του 8ης γραμμής. Επομένως, η avg δεν θα έχει λάβει την τελική (σωστή) τιμή της.

Αυτή είναι μία περίπτωση όπου μία οντότητα (νήμα T_1) δεν πρέπει να συνεχίσει με την εκτέλεση του κώδικά της μέχρι να σιγουρευτεί ότι κάποιες άλλες οντότητες (νήμα T_2) έχουν φτάσει σε κάποιο προκαθορισμένο σημείο του δικού τους κώδικα. Είναι λοιπόν απαραίτητες, εκτός από τον αμοιβαίο αποκλεισμό, και επιπλέον λειτουργίες για το *συγχρονισμό* (synchronization) των οντοτήτων.[2] Οι οντότητες, στα κατάλληλα σημεία του κώδικα, καλούν τις δομές συγχρονισμού και σταματάει αμέσως η εκτέλεσή τους μέχρι όλες οι υπόλοιπες να καλέσουν ανάλογες δομές συγχρονισμού. Όταν γίνει αυτό, μπορούν αμέσως όλες να συνεχίσουν την εκτέλεσή τους.

4.3 Νήματα και κοινόχρηστες μεταβλητές

Ένα νήμα, όπως είπαμε, είναι ουσιαστικά ένας μετρητής προγράμματος και μία στοίβα. Τα δύο αυτά «συστατικά» είναι αρκετά για να έχουμε μία εκτελεστική οντότητα που εκτελεί, ανεξάρτητα από άλλες, τμήματα του κώδικα. Τα νήματα μίας διεργασίας εκτελούν μεν εντολές ανεξάρτητα μεταξύ τους, μοιράζονται δε τον ίδιο κώδικα, δηλαδή μπορούν να καλέσουν τις ίδιες συναρτήσεις. Αυτό που τα διαφοροποιεί είναι ότι το καθένα διαθέτει *ιδιωτική* στοίβα και επομένως οι τοπικές μεταβλητές, κατά την κλήση της κάθε συνάρτησης, αποθηκεύονται ξεχωριστά στο κάθε νήμα. Αυτό σημαίνει ότι όλες οι τοπικές μεταβλητές είναι επίσης και *ιδιωτικές* μεταβλητές για τα νήματα. Το αντίθετο, ακριβώς, συμβαίνει για τις καθολικές μεταβλητές. Οι μεταβλητές αυτές, όπως έχουμε πει, αποθηκεύονται σε ξεχωριστό χώρο στη μνήμη, και βρίσκονται εκεί μέχρι την ολοκλήρωση της διεργασίας. Αυτό σημαίνει ότι όσα νήματα και να δημιουργήσει μία διεργασία, όλα θα «βλέπουν» τον ίδιο ακριβώς χώρο και θα προσπελαύνουν τις ίδιες ακριβώς μεταβλητές. Επομένως, κάθε καθολική μεταβλητή είναι επίσης και *κοινόχρηστη* μεταξύ των νημάτων είτε το επιθυμούμε είτε

[2] Στη βιβλιογραφία πολλές φορές (κακώς) ο όρος «συγχρονισμός» συμπεριλαμβάνει και την έννοια του αμοιβαίου αποκλεισμού.

όχι. Συμπερασματικά, όταν προγραμματίζουμε με νήματα, έχουμε «αυτόματα» έτοιμο τον απαραίτητο μηχανισμό για τον ορισμό κοινόχρηστων μεταβλητών: όποια μεταβλητή είναι καθολική στο πρόγραμμά μας, είναι (υποχρεωτικά) και κοινόχρηστη μεταξύ των νημάτων που θα δημιουργήσουμε.

Έχουμε, επομένως, τον τρόπο για ορισμό κοινόχρηστων μεταβλητών. Πώς όμως δημιουργούμε ένα νήμα; Υπάρχουν πολλές υλοποιήσεις νημάτων, με εντελώς διαφορετικά χαρακτηριστικά μεταξύ τους. Εμείς θα χρησιμοποιήσουμε εδώ τα λεγόμενα νήματα POSIX, γνωστά επίσης και ως *Pthreads*. Τα νήματα αυτά αποτελούν καθολικό πρότυπο και παρέχονται βιβλιοθήκες που τα υλοποιούν σε όλα σχεδόν τα λειτουργικά συστήματα. Η διεπαφή για το πώς κάποιος δημιουργεί και διαχειρίζεται τα νήματα έχει καθοριστεί στο παγκόσμιο πρότυπο POSIX.1c και τις επεκτάσεις αυτού.

Μία διεργασία, στην εκκίνησή της, αποτελείται από ένα και μοναδικό νήμα, το οποίο το ονομάζουμε *αρχικό νήμα* της διεργασίας, και το οποίο αναλαμβάνει την εκτέλεση του κώδικα της `main()`. Η δημιουργία ενός νέου νήματος γίνεται με την κλήση `pthread_create()`, και καθορίζεται μία συνάρτηση που θα εκτελέσει το νήμα. Όταν η συνάρτηση αυτή επιστρέψει, το νήμα ολοκληρώνει την εκτέλεσή του και καταστρέφεται από το σύστημα. Η κλήση έχει ως εξής:

```
pthread_create(thrid, attrs, funcptr, arg);
```

όπου:

- `thrid` είναι δείκτης σε μεταβλητή τύπου `pthread_t`, όπου θα αποθηκευτεί το *αναγνωριστικό* του νέου νήματος,
- `attrs` είναι δείκτης σε μία μεταβλητή τύπου `pthread_attr_t`, με τα χαρακτηριστικά (attributes) που θα θέλαμε να έχει το νήμα,
- `funcptr` είναι δείκτης στη συνάρτηση που πρέπει να εκτελέσει το νήμα, και
- `arg` είναι το όρισμα που θα δοθεί στη συνάρτηση του νήματος, και είναι πάντα δείκτης `void`.

Μόλις δημιουργηθεί το νήμα, το μόνο που κάνει είναι να καλέσει άμεσα την καθορισμένη συνάρτηση (`funcptr`), με το όρισμα που δηλώσαμε (`arg`). Το νήμα «ζει» όσο εκτελεί τη συνάρτηση· όταν η συνάρτηση επιστρέφει, τερματίζεται αυτόματα και το νήμα.

Σε κάθε νήμα αποδίδεται ένα μοναδικό αναγνωριστικό, τύπου `pthread_t`, το οποίο επιστρέφεται στο `thrid`, προκειμένου να το γνωρίζει αυτός που δημιουργεί το νήμα (ο «γονέας» του νήματος). Το ίδιο το νήμα, όταν εκτελείται μπορεί να μάθει το αναγνωριστικό του, από την τιμή που επιστρέφει η συνάρτηση `pthread_self()`:

```
pthread_t myid = pthread_self();
```

Δυστυχώς, δεν πρέπει να γίνεται καμία υπόθεση για το αναγνωριστικό αυτό, αν και αρκετές φορές ο τύπος του είναι ακέραιος. Ακόμα κι έτσι, όμως, τα αναγνωριστικά των νημάτων

```
1  #include <pthread.h>
2
3  char *strings[10];           /* 10 pointers to strings (shared) */
4
5  void *capitalize(void *str) {
6    ...                        /* Capitalize all letters */
7  }
8
9  int main() {
10   int       i;
11   pthread_t thrid[10];
12
13   read_strings(strings);     /* Get the strings somehow */
14   for (i = 1; i < 10; i++)   /* Create 9 threads to process them */
15     pthread_create(&thrid[i], NULL, capitalize, (void *) string[i]);
16   capitalize((void *) string[0]);  /* Participate, too! */
17   ...
18   for (i = 1; i < 10; i++)   /* Wait for the 9 threads to finish */
19     pthread_join(thrid[i], NULL);
20   ...
21 }
```

Πρόγραμμα 4.2 *Δημιουργία 9 νημάτων από το αρχικό νήμα και αναμονή για τον τερματισμό τους.*

είναι τυχαίοι αριθμοί. Αυτό δεν βολεύει καθόλου στο σχεδιασμό των προγραμμάτων, όπου επιθυμούμε να διαχειριζόμαστε N νήματα σαν να αριθμούνται ακολουθιακά, 0, 1, 2, ..., $N-1$. Σε αυτή την περίπτωση, εκτός από το αναγνωριστικό, μπορούμε να δώσουμε εμείς σε κάθε νήμα μία δική μας *ταυτότητα*, όπως θα δούμε παρακάτω.

Κατά τη δημιουργία του νήματος, μπορούμε να καθορίσουμε κάποια επιθυμητά χαρακτηριστικά του, όπως για παράδειγμα το μέγεθος της στοίβας του, τα οποία όμως ξεφεύγουν από τα ενδιαφέροντά μας εδώ. Αυτό γίνεται μέσω της παραμέτρου `attrs` της `pthread_create()`. Εκτός από εξειδικευμένες περιπτώσεις, όπου απαιτείται λεπτομερής έλεγχος της λειτουργίας των νημάτων, η συνήθης τακτική είναι να δίνεται η τιμή `NULL`, η οποία αφήνει το σύστημα να δημιουργήσει το νήμα με τα δικά του, προκαθορισμένα χαρακτηριστικά.

Στο Πρόγρ. 4.2 δίνεται ένα απλό, αλλά πλήρες παράδειγμα δημιουργίας νημάτων όπου το καθένα υπολογίζει κάτι διαφορετικό. Το αρχικό νήμα, που εκτελεί την `main()`, συγκεντρώνει 10 συμβολοσειρές, και στη συνέχεια, δημιουργεί ένα νέο νήμα για κάθε μία συμβολοσειρά (γραμμές 13–14). Όπως φαίνεται στη γραμμή 14, τα νήματα όλα θα εκτελέσουν τη συνάρτηση `capitalize()`, όμως με διαφορετικό όρισμα το καθένα. Το i-οστό νήμα θα λάβει ως όρισμα έναν δείκτη στην i-οστή συμβολοσειρά. Το ίδιο το αρχικό νήμα θα συνεχίσει την εκτέλεσή του στη γραμμή 15 και θα καλέσει την ίδια συνάρτηση για τη συμ-

βολοσειρά 0. Επομένως, μαζί με το αρχικό, υπάρχουν 10 νήματα συνολικά, και όλα εκτελούν την ίδια συνάρτηση, αλλά με χρήση της ιδιωτικής στοίβας του καθενός. Αν υπήρχαν 10 πυρήνες διαθέσιμοι, το πιθανότερο είναι όλα τα νήματα να δρομολογηθούν ταυτόχρονα, σε διαφορετικό πυρήνα το καθένα, και επομένως να έχουμε πλήρως παράλληλη εκτέλεση. Από την άλλη, αν υπάρχουν πολύ λιγότεροι πυρήνες, τα 10 αυτά νήματα θα δρομολογηθούν τμηματικά, με κάποια σειρά, η οποία όμως είναι εντελώς τυχαία. Για παράδειγμα, δεν μπορεί κάποιος να γνωρίζει αν το αρχικό νήμα θα φτάσει πρώτο στη γραμμή 15 ή θα προλάβει πρώτα κάποιο από τα νήματα-παιδιά του να εκτελεστεί και ίσως, να ολοκληρωθεί.

Παρότι η εκτέλεση των νημάτων γίνεται ανεξάρτητα από αυτή του γονέα, σχεδόν σε όλες τις περιπτώσεις (και, πάντως, σίγουρα στην προγραμματιστική στρατηγική fork-join που περιγράψαμε πριν), ο γονέας θα πρέπει να περιμένει τα παιδιά του να ολοκληρώσουν το έργο τους προκειμένου να συνεχίσει σε επόμενες λειτουργίες. Η κλήση `pthread_join()` παρέχει τη δυνατότητα αυτή στο γονέα, όπως φαίνεται στις γραμμές 17–18. Η κλήση αυτή κάνει τον γονέα να περιμένει τον τερματισμό του νήματος-παιδιού, το αναγνωριστικό του οποίου δίνεται ως πρώτη παράμετρος. Η δεύτερη παράμετρος είναι ένας δείκτης σε δείκτη `void` (δηλαδή `void **`), και η σημασία της αναλύεται αμέσως μετά.

Προσέξτε τον ορισμό της συνάρτησης `capitalize()`. Η συνάρτηση που δίνεται σε ένα νήμα να εκτελέσει, πρέπει να δέχεται ακριβώς ένα όρισμα, τύπου δείκτη `void`, αλλά και να *επιστρέφει* πάντα ένα δείκτη `void`. Η τιμή του δείκτη μπορεί να χρησιμοποιηθεί από το νήμα, ώστε να επιστρέψει κάτι στον γονέα του, εφόσον χρειάζεται. Ο μόνος τρόπος να μάθει ο γονέας την τιμή επιστροφής είναι μέσω της `pthread_join()`, και συγκεκριμένα από το δεύτερο όρισμά της. Παρέχοντας έναν δείκτη προς μία μεταβλητή τύπου `void *`, κατά τον τερματισμό του νήματος-παιδιού, αντιγράφεται εκεί η τιμή που θα επιστρέψει η συνάρτηση που εκτέλεσε το νήμα. Στο συγκεκριμένο παράδειγμα, η τιμή `NULL` σημαίνει ότι ο γονέας δεν ενδιαφέρεται να μάθει τι επέστρεψε το παιδί του.

Πριν ολοκληρώσουμε την ενότητα αυτή, πρέπει να σημειώσουμε ότι υπάρχει και δεύτερος τρόπος να τερματιστεί ένα νήμα, εκτός από το να επιστρέψει η συνάρτηση που του δόθηκε προς εκτέλεση. Ο τρόπος αυτός είναι να κληθεί από το ίδιο το νήμα η συνάρτηση:

```
pthread_exit(retval);
```

Η παράμετρος `retval` είναι δείκτης `void` και αποτελεί την τιμή επιστροφής προς τον γονέα. Η συνάρτηση έχει ως αποτέλεσμα τον άμεσο τερματισμό του νήματος που την καλεί. Η κλήση της μπορεί να γίνει σε οποιοδήποτε σημείο του κώδικα επιθυμούμε.

Από το Πρόγρ. 4.2, φαίνεται ότι τα προγράμματα που κάνουν χρήση των νημάτων POSIX πρέπει να συμπεριλαμβάνουν το αρχείο επικεφαλίδων `<pthread.h>`. Επιπλέον, η μετάφραση των προγραμμάτων θα πρέπει να γίνεται με ορισμό της σταθεράς `_REENTRANT` και συνένωση με τη βιβλιοθήκη (library) των Pthreads. Υποθέτοντας ότι το όνομα του προγράμματος είναι `prog.c`, και θεωρώντας ότι ο μεταφραστής είναι ο γνωστός GCC της GNU, η μετάφραση γίνεται ως εξής:

```
% gcc prog.c -D_REENTRANT -lpthread
```

Στα περισσότερα συστήματα, όμως, αρκεί απλά το όρισμα -pthread:

```
% gcc -pthread prog.c
```

4.4 Αμοιβαίος αποκλεισμός με κλειδαριές

Όπως είδαμε και στην εισαγωγική ενότητα, είναι απαραίτητη η ύπαρξη προγραμματιστικών δομών για την επίτευξη αμοιβαίου αποκλεισμού μεταξύ των οντοτήτων εκτέλεσης. Από τη στιγμή που υπάρχουν νήματα τα οποία τροποποιούν μία κοινόχρηστη μεταβλητή, υπάρχει και η ανάγκη για αμοιβαίο αποκλεισμό. Σε κάθε νήμα, τα τμήματα του κώδικα που τροποποιούν την κοινή μεταβλητή είναι γνωστά ως *κρίσιμες περιοχές* (critical sections).

Προκειμένου να αποφύγουμε την ταυτόχρονη τροποποίηση μίας κοινής μεταβλητής από δύο ή παραπάνω νήματα, αρκεί να επιτρέπουμε μόνο σε ένα από αυτά τη φορά να εκτελεί κώδικα σε κρίσιμη περιοχή του. Επομένως, όταν ένα νήμα βρίσκεται σε κρίσιμη περιοχή, κανένα άλλο δεν μπορεί να εισέλθει σε αντίστοιχη κρίσιμη περιοχή, και θα πρέπει να περιμένει μέχρι το πρώτο νήμα να ολοκληρώσει τις κρίσιμες εντολές του. Αυτό επιτυγχάνεται με τη χρήση δομών αμοιβαίου αποκλεισμού ακριβώς *πριν* και ακριβώς *μετά* την κρίσιμη περιοχή κάθε νήματος.

Δεν υπάρχει μόνο ένας τύπος προγραμματιστικών δομών για την επίτευξη αμοιβαίου αποκλεισμού. Ενδεικτικά αναφέρουμε τις *κλειδαριές* (locks), τους *σημαφόρους* (semaphores), και τα *monitors*. Τα τελευταία είναι ένας αντικειμενοστραφής μηχανισμός υψηλού επιπέδου (σε σχέση με τις κλειδαριές και τους σηματοφόρους) και έχουν χρησιμοποιηθεί στις παλαιότερες γλώσσες Concurrent Pascal και Modula P. Δεν είναι όμως δημοφιλή (ούτε και ιδιαίτερα γρήγορα) και δεν θα μας απασχολήσουν παραπάνω. Οι σημαφόροι είναι ένας κλασικός μηχανισμός αμοιβαίου αποκλεισμού, ο οποίος όμως χρησιμοποιείται περισσότερο όταν οι οντότητες εκτέλεσης είναι διεργασίες και σε γενικές γραμμές θεωρείται σχετικά "αργός". Επίσης, πρόσφατα έχουν προταθεί νέοι μηχανισμοί όπως η *μνήμη δοσοληψιών* (transactional memory) που πιθανώς αποδειχτούν επιτυχημένοι σε κάποιες κατηγορίες εφαρμογών.

Οι κλειδαριές είναι, κατά πολύ, η απλούστερη και πιο ευρέως χρησιμοποιούμενη μέθοδος αμοιβαίου αποκλεισμού στον παράλληλο προγραμματισμό, αν και ανάλογα με την υλοποίηση του μηχανισμού τους, μπορεί να μην είναι πάντα η αποδοτικότερη. Μία κλειδαριά *είναι απλά μία κοινόχρηστη μεταβλητή*, η οποία, όμως, μπορεί να πάρει μόνο δύο τιμές: «κλειδωμένη» ή «ξεκλείδωτη». Το κλείδωμα και το ξεκλείδωμα μίας κλειδαριάς γίνεται με κλήση σε κατάλληλες συναρτήσεις, η λειτουργία των οποίων παρουσιάζεται παρακάτω. Για να επιτύχουμε αμοιβαίο αποκλεισμό, πρέπει να τηρήσουμε το εξής πρωτόκολλο:

- Προκειμένου ένα νήμα να εισέλθει στην κρίσιμη περιοχή του, πρέπει να κλειδώσει την κλειδαριά.
- Όταν εξέλθει από την κρίσιμη περιοχή του, πρέπει να ξεκλειδώσει την κλειδαριά.

Επομένως, σε όσα κομμάτια του κώδικα απαιτείται αμοιβαίος αποκλεισμός, θα πρέπει να τα "ντύνουμε" με κλήσεις κλειδώματος, στην αρχή τους, και ξεκλειδώματος, στο τέλος τους.

Θεωρώντας ότι αρχικά η κλειδαριά είναι ξεκλείδωτη, όποιο νήμα επιθυμεί να εισέλθει στην κρίσιμη περιοχή του, την κλειδώνει και εκτελεί τον "επικίνδυνο" κώδικα. Αν, εντωμεταξύ, εμφανιστεί ένα οποιοδήποτε άλλο νήμα, το οποίο θέλει κι αυτό να εκτελέσει μία κρίσιμη περιοχή του, τότε κατά την κλήση κλειδώματος θα διαπιστωθεί ότι η κλειδαριά είναι ήδη κλειδωμένη. Στην περίπτωση αυτή *η κλήση κλειδώματος δεν επιστρέφει* ("μπλοκάρει"). Επομένως, δεν μπορεί το νήμα αυτό να συνεχίσει και να εισέλθει στην κρίσιμη περιοχή του· έχουμε, επομένως, επίτευξη αμοιβαίου αποκλεισμού.

Το νήμα που κλείδωσε την κλειδαριά, μόλις τελειώσει την εκτέλεση της κρίσιμης περιοχής, καλεί τη συνάρτηση ξεκλειδώματος, η οποία και θα απελευθερώσει την κλειδαριά. Έτσι, κάποιο από τα νήματα που ήταν μπλοκαρισμένα επειδή είχαν βρει κλειδωμένη την κλειδαριά, θα έχει την ευκαιρία να την ξεκλειδώσει, και να μπει στην κρίσιμη περιοχή του.

Η βιβλιοθήκη των νημάτων posix προσφέρει κλειδαριές για αμοιβαίο αποκλεισμό, υπό το όνομα *mutexes*. Ένα mutex είναι μία μεταβλητή τύπου `pthread_mutex_t`, όπως στην παρακάτω δήλωση:

```
pthread_mutex_t mx;
```

Οι βασικές κλήσεις που επηρεάζουν την κλειδαριά είναι οι εξής:

```
pthread_mutex_init(&mx, attrs);
pthread_mutex_lock(&mx);
pthread_mutex_unlock(&mx);
```

Όπως είναι φανερό, η δεύτερη και η τρίτη κλήση κλειδώνουν και ξεκλειδώνουν, αντίστοιχα, την κλειδαριά, η οποία περνιέται με αναφορά. Η πρώτη κλήση είναι απαραίτητη, πριν κάποιος κάνει οποιαδήποτε χρήση μίας μεταβλητή τύπου mutex, και αναλαμβάνει την αρχικοποίησή της. Το `attrs` είναι δείκτης σε μία μεταβλητή τύπου `pthread_mutex_attr_t`, με διάφορα χαρακτηριστικά (attributes) που θα θέλαμε να έχει η κλειδαριά. Εμείς εδώ, θα περνάμε πάντα την τιμή `NULL`, ώστε οι κλειδαριές να έχουν τα συνήθη χαρακτηριστικά που έχουν οριστεί στο σύστημα και επομένως, η κλήση αρχικοποίησης θα έχει πάντα τη μορφή `pthread_mutex_init(&mx, NULL)`. Πολλές φορές είναι πιο βολικό να αρχικοποιούμε μία κλειδαριά τη στιγμή που τη δηλώνουμε. Για τον λόγο αυτό, υπάρχει η δυνατότητα να χρησιμοποιηθεί και ο λεγόμενος στατικός αρχικοποιητής, ως εξής:

```
pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER; // Declare+initialize
```

Όπως είδαμε στην Ενότητα 4.2, ο κώδικας του Σχήματος 4.1, ο οποίος υπολογίζει τον μέσο όρο 10 αριθμών με 2 νήματα, υποφέρει λόγω έλλειψης αμοιβαίου αποκλεισμού κατά

```
int             A[10];
double          avg = 0.0;
pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;

void *threadfunc1(void *arg) {
  int i, q1 = 0;

  for (i = 0; i < 5; i++)
    q1 += A[i];
  pthread_mutex_lock(&mx);
  avg = avg + (q1 / 10.0);       /* critical section */
  pthread_mutex_unlock(&mx);
  ...
}

void *threadfunc2(void *arg) {
  int i, q2 = 0;

  for (i = 5; i < 10; i++)
    q2 += A[i];
  pthread_mutex_lock(&mx);
  avg = avg + (q2 / 10.0);       /* critical section */
  pthread_mutex_unlock(&mx);
  printf("avg = %lf\n", avg);    /* show result */
  ...
}
```

Πρόγραμμα 4.3 *Οι συναρτήσεις που εκτελούν δύο νήματα τα οποία συνεργάζονται για τον υπολογισμό του μέσου όρου 10 στοιχείων.*

την καταχώρηση στη μεταβλητή avg. Η ορθή εκδοχή δίνεται στο Πρόγρ. 4.3, όπου κάνουμε χρήση των κλειδαριών που προσφέρουν τα Pthreads. Προσέξτε ότι και οι τρεις καθολικές μεταβλητές, συμπεριλαμβανομένης και της κλειδαριάς mx, είναι εξ ορισμού κοινόχρηστες. Η κλειδαριά είναι αρχικοποιημένη στατικά. Το πρώτο νήμα υποτίθεται ότι εκτελεί τη συνάρτηση threadfunc1() και το δεύτερο την threadfunc2(). Και στις δύο συναρτήσεις, έχουμε φροντίσει να προστατέψουμε τον κρίσιμο κώδικα (δηλαδή την τροποποίηση της τιμής της κοινόχρηστης avg στις γραμμές 11 και 22) από πριν, με κλήση για κλείδωμα της mx, και αμέσως μετά, με κλήση για ξεκλείδωμα. Το τελικό αποτέλεσμα εκτυπώνεται από το δεύτερο νήμα.[3]

Ο αμοιβαίος αποκλεισμός είναι ευθύνη του προγραμματιστή. Αυτός θα πρέπει να αναγνωρίσει τις κρίσιμες περιοχές και σε αυτόν επαφίεται η χρήση ή όχι κατάλληλων μηχανισμών, όπως οι κλειδαριές. Η χρήση των μηχανισμών αυτών θα πρέπει να είναι προσεκτική,

[3]Σημειώνουμε ότι το Πρόγρ. 4.3 έχει ένα ακόμη άλλο λογικό σφάλμα, το οποίο θα δούμε παρακάτω.

καθώς μπορεί να προκαλέσει απρόσμενα προβλήματα, όπως π.χ. το *αδιέξοδο* (deadlock) όπου κάποια νήματα περιμένουν κάτι το οποίο δεν πρόκειται να συμβεί ποτέ.[4]

4.5 Συγχρονισμός

Κατά την παράλληλη εκτέλεση των νημάτων, εκτός από την ανάγκη για αμοιβαίο αποκλεισμό, υπάρχει πολλές φορές και η ανάγκη για *συγχρονισμό* (synchronization), δηλαδή για αναμονή των νημάτων σε προκαθορισμένα σημεία του κώδικα έως ότου, ακόμα και τα πιο αργά νήματα να φτάσουν εκεί ή κάποιο σημαντικό γεγονός να συμβεί. Πολλά παράλληλα προγράμματα που έχουν γραφτεί στο μοντέλο κοινόχρηστου χώρου διευθύνσεων μπορούν να χωριστούν σε λογικές φάσεις που εκτελούνται η μία μετά την άλλη. Συνήθως, αυτό συμβαίνει όταν, για να υπολογιστούν κάποια μεγέθη, θα πρέπει πρώτα να έχει ολοκληρωθεί ο υπολογισμός κάποιων άλλων. Για παράδειγμα, για να λυθεί ένα σύστημα εξισώσεων $Ax = b$, μία μέθοδος είναι να γίνει πρώτα ο πίνακας A τριγωνικός (π.χ. με απαλοιφή Gauss) και αφού γίνει αυτό, το άγνωστο διάνυσμα x να υπολογιστεί με την προς-τα-πίσω αντικατάσταση (back-substitution). Ας υποθέσουμε ότι κάθε νήμα αναλαμβάνει να συνεισφέρει τόσο στην απαλοιφή όσο και στην προς-τα-πίσω αντικατάσταση. Θα πρέπει να είναι φανερό ότι, αν ένα νήμα τελειώσει το τμήμα της απαλοιφής που ανέλαβε, *δεν* θα πρέπει να ξεκινήσει την προς-τα-πίσω αντικατάσταση. Ο λόγος είναι ότι η απαλοιφή μπορεί να μην έχει ολοκληρωθεί πλήρως εξαιτίας πιθανής αργοπορίας των υπολοίπων νημάτων και επομένως, το αποτέλεσμα θα μπορούσε να βγει λανθασμένο.

Σε τέτοιες περιπτώσεις, το αποτέλεσμα των υπολογισμών εξαρτάται από τη συμπεριφορά των νημάτων και συγκεκριμένα από τη σχετική τους *ταχύτητα*. Αν όλα τα νήματα ολοκληρώσουν τη φάση της απαλοιφής ταυτόχρονα (κάτι που δεν το εγγυάται κανείς), δεν υπάρχει κανένα πρόβλημα. Αν όμως κάποιο νήμα είναι ταχύτερο από τα υπόλοιπα, τότε η είσοδός του στη φάση της προς-τα-πίσω αντικατάστασης θα δώσει λανθασμένο αποτέλεσμα, μιας και ο πίνακας A δεν θα έχει λάβει την τελική τριγωνική μορφή του. Όταν το τελικό αποτέλεσμα εξαρτάται από τις σχετικές ταχύτητες εκτέλεσης των νημάτων, λέμε ότι επικρατούν *συνθήκες ανταγωνισμού* (race conditions), κάτι το οποίο είναι φανερά ανεπιθύμητο.

Η λύση στα προβλήματά μας είναι ο συγχρονισμός των νημάτων. Τα νήματα που μας ενδιαφέρουν αναγκάζονται να περιμένουν σε κάποιο σημείο του κώδικα, έως ότου ολοκληρώσουν τους υπολογισμούς τους και τα υπόλοιπα. Ένα καλό σημείο συγχρονισμού στο παραπάνω παράδειγμα του συστήματος εξισώσεων είναι εκεί που τελειώνει η πρώτη φάση των υπολογισμών (απαλοιφή). Κανένα νήμα που φτάνει εκεί δεν επιτρέπεται να εισέλθει στη δεύτερη φάση (αντικατάσταση προς τα πίσω), μέχρις ότου όλα τα άλλα νήματα

[4] Για παράδειγμα, προσπαθήστε να απαντήσετε το Πρόβλημα 4.1.

ολοκληρώσουν την πρώτη φάση. Προσέξτε ότι ο συγχρονισμός είναι διαφορετική έννοια από τον αμοιβαίο αποκλεισμό, αν και πολλοί θεωρούν και τον αμοιβαίο αποκλεισμό ως είδος συγχρονισμού.

Η βιβλιοθήκη των Pthreads προσφέρει εγγενώς έναν μηχανισμό αναμονής για κάποιο γεγονός, τις ονομαζόμενες *μεταβλητές συνθήκης* (condition variables). Επιπλέον, στα περισσότερα συστήματα προσφέρεται και ο πιο δημοφιλής μηχανισμός των *κλήσεων φραγής* (barriers).

4.5.1 Μεταβλητές συνθήκης

Η μεταβλητή συνθήκης είναι μία κοινόχρηστη προγραμματιστική δομή η οποία σταματά προσωρινά («μπλοκάρει») κάποιο νήμα μέχρι κάποια συγκεκριμένη συνθήκη να γίνει αληθής. Επειδή το νήμα θα είναι μπλοκαρισμένο, για να γίνει η συνθήκη αληθής θα πρέπει να την επηρεάσουν τα υπόλοιπα νήματα, και επομένως, στη συνθήκη υποχρεωτικά θα πρέπει να εμπλέκεται και κάποια κοινόχρηστη μεταβλητή. Αυτό, με τη σειρά του, δημιουργεί την ανάγκη για αμοιβαίο αποκλεισμό. Έτσι, οι μεταβλητές συνθήκης χρησιμοποιούνται πάντα σε συνδυασμό με μία κλειδαριά.

Οι μεταβλητές συνθήκης, είναι τύπου `pthread_cond_t`, και όπως και οι κλειδαριές, πρέπει να αρχικοποιηθούν πριν χρησιμοποιηθούν από τα νήματα. Η αρχικοποίηση γίνεται είτε μέσω συνάρτησης είτε στατικά κατά τη δήλωση, όπως στον παρακάτω κώδικα:

```
pthread_cond_t cv1, cv2 = PTHREAD_COND_INITIALIZER;
pthread_cond_init(&cv1, NULL);
```

Όπως και στις κλειδαριές, η συνάρτηση αρχικοποίησης παίρνει ως δεύτερο όρισμα έναν δείκτη σε χαρακτηριστικά που θα θέλαμε να έχει η μεταβλητή συνθήκης. Εμείς εδώ θα περνάμε πάντα NULL, ώστε να χρησιμοποιηθούν τα τυπικά χαρακτηριστικά που έχει ορίσει το σύστημα.

Αφού αρχικοποιηθεί, τρεις είναι οι κλήσεις που σχετίζονται με τον συγχρονισμό βάσει μίας μεταβλητής συνθήκης. Ένα νήμα μπορεί να εισέλθει στην κατάσταση αναμονής, καλώντας τη συνάρτηση `pthread_cond_wait()`, που θα δούμε παρακάτω. Κάποιο νήμα μπορεί να «ξυπνήσει» (ή να «σηματοδοτήσει», όπως λέγεται) άλλο νήμα που περιμένει επάνω σε μία μεταβλητή συνθήκης, καλώντας τη συνάρτηση: `pthread_cond_signal()`. Η κλήση αυτή ενεργοποιεί *ένα* (τυχαίο) από όλα τα νήματα που περιμένουν. Εάν σκοπός μας είναι να ενεργοποιήσουμε *όλα* τα εν αναμονή νήματα, τότε θα πρέπει να χρησιμοποιηθεί η συνάρτηση `pthread_cond_broadcast()`.

Η κλήση που θέτει ένα νήμα σε κατάσταση αναμονής συντάσσεται όπως παρακάτω:

```
pthread_cond_wait(&condvar, &lock);
```

όπου `lock` είναι το mutex που, όπως είπαμε, πρέπει να χρησιμοποιείται μαζί με την μεταβλητή συνθήκης, και το οποίο *πρέπει να έχει κλειδωθεί πριν την κλήση* στην `pthread_cond_wait()`.

```
1  pthread_cond_t  condvar = PTHREAD_COND_INITIALIZER;
2  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
3  int             flag = FALSE;
4
5  void *threadA(void *arg) {
6    ...
7    pthread_mutex_lock(&lock);
8    while (flag != TRUE)      /* waiting for flag to become true */
9      pthread_cond_wait(&condvar, &lock);
10   pthread_mutex_unlock(&lock);
11   ...
12 }
13
14 void *threadB(void *arg) {
15   ...
16   pthread_mutex_lock(&lock);
17   flag = TRUE;
18   pthread_cond_signal(&condvar);
19   pthread_mutex_unlock(&lock);
20   ...
21 }
```

Πρόγραμμα 4.4 *Δύο νήματα που συγχρονίζονται μέσω μεταβλητής συνθήκης.*

Η συνάρτηση αυτή, καθώς θέτει το νήμα σε κατάσταση αναμονής, φροντίζει επίσης να ξεκλειδώσει την κλειδαριά (ώστε να είναι ελεύθερη για τα άλλα νήματα). Όταν, όμως, το νήμα είναι έτοιμο να ξυπνήσει για κάποιο λόγο, τότε το σύστημα φροντίζει να κλειδώσει την κλειδαριά, ώστε κατά την επανενεργοποίησή του το νήμα να είναι πάλι μέσα στην κρίσιμη περιοχή του.

Η τυπική χρήση των μεταβλητών συνθήκης δίνεται στο Πρόγρ. 4.4. Το νήμα A το οποίο εκτελεί την threadA() θέλει να περιμένει μέχρι η μεταβλητή flag να γίνει TRUE (αυτή είναι η συνθήκη μας). Για να το καταφέρει αυτό, πρέπει επιπλέον να χρησιμοποιήσει μία μεταβλητή συνθήκης (condvar) και μία κλειδαριά (lock). Προκειμένου να ελέγξει την τιμή του flag, κλειδώνει πρώτα την κλειδαριά (γραμμή 7). Στη συνέχεια, και εφόσον το flag δεν είναι TRUE μπλοκάρει μέσω της pthread_cond_wait() (γραμμή 9), η οποία, όπως είπαμε, θα ξεκλειδώσει αυτόματα την κλειδαριά. Αν το flag είναι TRUE το νήμα δεν (ξανα)κοιμάται, ξεκλειδώνει την κλειδαριά και συνεχίζει την εκτέλεσή του.

Το νήμα B κάποια στιγμή θα κλειδώσει την ανοιχτή κλειδαριά (γραμμή 16), θα θέσει το flag στην τιμή TRUE και θα σηματοδοτήσει το ξύπνημα του μπλοκαρισμένου νήματος (γραμμή 18). Το ξεκλείδωμα στην γραμμή 19 είναι απαραίτητο διότι, όπως είπαμε παραπάνω, το νήμα A ενεργοποιείται πλήρως μόνο αφού κατορθώσει να πάρει την κλειδαριά μετά το ξύπνημά του.

```
/* N threads call this */
void *producers(void *arg) {
  ...
  pthread_mutex_lock(&lock);
  count++;
  if (count == N)
    pthread_cond_signal(&condvar);
  pthread_mutex_unlock(&lock);
  ...
}

void *consumer(void *arg) {
  ...
  pthread_mutex_lock(&lock);
  while (count < N)
    pthread_cond_wait(&condvar,&lock);
  pthread_mutex_unlock(&lock);
  ...
}
```

```
void *producer(void *arg) {
  ...
  pthread_mutex_lock(&lock);
  count++;
  if (count == N)
    pthread_cond_broadcast(&condvar);
  pthread_mutex_unlock(&lock);
  ...
}

/* N threads call this */
void *consumers(void *arg) {
  ...
  pthread_mutex_lock(&lock);
  while (count < N)
    pthread_cond_wait(&condvar,&lock);
  pthread_mutex_unlock(&lock);
  ...
}
```

Πρόγραμμα 4.5 *Ένα νήμα περιμένει μία συνθήκη που την επηρεάζουν N άλλα νήματα (αριστερά) και N νήματα περιμένουν μία συνθήκη που την επηρεάζει ένα άλλο νήμα (δεξιά).*

Μία σημαντική λεπτομέρεια, που πρέπει να επισημάνουμε, είναι ότι τα σήματα που αποστέλλονται με την `pthread_cond_signal()` και την `pthread_cond_broadcast()`, προκειμένου να ξυπνήσουν τυχόν μπλοκαρισμένα νήματα *χάνονται*, αν δεν υπάρχει κανείς να τα παραλάβει (δηλαδή αν δεν υπάρχει νήμα σε κατάσταση αναμονής). Επομένως, αν ένα νήμα στείλει ένα σήμα και *μετά* μπλοκάρει ένα άλλο νήμα, πιθανώς το δεύτερο να μην ξυπνήσει ποτέ. Αυτός είναι και ο λόγος που χρησιμοποιούμε την κλειδαριά, όχι μόνο στο νήμα που θα κοιμηθεί αλλά και σε αυτό που θα προκαλέσει το ξύπνημα[5].

Άλλες δύο συνήθεις περιπτώσεις συγχρονισμού σε εφαρμογές, όπου τα νήματα έχουν σχέσεις παραγωγού-καταναλωτή, δίνονται στο Πρόγρ. 4.5. Στην πρώτη (αριστερά), ένα νήμα ("καταναλωτής") περιμένει τον μετρητή `count` να πάρει την τιμή N. Η τιμή του αυξάνει από καθένα από τα υπόλοιπα N νήματα ("παραγωγοί") που υπάρχουν στο σύστημα. Μόλις ένα από τα νήματα αυτά διαπιστώσει ότι η τιμή του `count` έγινε ίση με N, σηματοδοτεί το (πιθανώς) μπλοκαρισμένο νήμα-καταναλωτή. Στην άλλη περίπτωση (Πρόγρ. 4.5, δεξιά), N νήματα περιμένουν μία συνθήκη η οποία επηρεάζεται από ένα άλλο νήμα. Όταν το τελευταίο δει ότι ο μετρητής `count` πάρει την τιμή N, τότε ξυπνάει όλα τα υπόλοιπα νήματα, με χρήση της `pthread_cond_broadcast()`.

[5]Προσπαθήστε να απαντήσετε στο Πρόβλημα 4.3.

4.5.2 Κλήσεις φραγής (barriers)

Αν και οι μεταβλητές συνθήκης είναι ένας ισχυρός και ευέλικτος μηχανισμός, θεωρείται σχετικά πρωτόγονος στον παράλληλο προγραμματισμό, και δεν χρησιμοποιείται τόσο συχνά. Ο πιο ευρέως χρησιμοποιούμενος μηχανισμός συγχρονισμού είναι οι λεγόμενες *κλήσεις φραγής*, γνωστές και ως `barriers`.

Ο μηχανισμός των barriers είναι εξαιρετικά απλός στη χρήση του: Υπάρχει μία συνάρτηση αναμονής την οποία καλούν τα νήματα στα κατάλληλα σημεία. Ως αποτέλεσμα, αναστέλλεται η εκτέλεση κάθε νήματος που την καλεί, έως ότου *όλα* τα νήματα καλέσουν την ίδια συνάρτηση. Όταν συμβεί αυτό, αρχίζουν πάλι όλα μαζί την εκτέλεσή τους. Με αυτόν τον τρόπο επιτυγχάνεται συγχρονισμός, σε όποιο σημείο του κώδικα απαιτείται. Όπως και με τον αμοιβαίο αποκλεισμό, έτσι και με το συγχρονισμό, είναι υποχρέωση του προγραμματιστή να ανακαλύψει πότε κάτι τέτοιο είναι αναγκαίο και σε ποιο σημείο πρέπει να τοποθετηθεί.

Η βιβλιοθήκη των Pthreads, με τις επεκτάσεις για τα λεγόμενα «προχωρημένα» νήματα «πραγματικού χρόνου» (που, πλέον, υπάρχουν σε όλα τα συμβατικά συστήματα), παρέχει κλήσεις φραγής. Ένα barrier είναι μία μεταβλητή τύπου `pthread_barrier_t`, και θα πρέπει να αρχικοποιηθεί πριν τη χρήση του ως εξής:

```
pthread_barrier_t bar;
pthread_barrier_init(&bar, NULL, N);
```

όπου ως δεύτερο όρισμα δίνουμε έναν δείκτη σε `pthread_barrierattr_t`, με χαρακτηριστικά που θα θέλαμε να έχει το barrier. Εμείς εδώ θα περνάμε πάντα NULL, ώστε να χρησιμοποιηθούν τα τυπικά χαρακτηριστικά που έχει ορίσει το σύστημα. Το τρίτο όρισμα είναι *το πλήθος των νημάτων που θα συμμετέχουν*. Με την κλήση:

```
pthread_barrier_wait(&bar);
```

ένα νήμα εισέρχεται στο barrier και αναστέλλει τη λειτουργία του. Θα επανεκκινηθεί όταν ακριβώς N νήματα καλέσουν την `pthread_barrier_wait()`, όπου το N είναι η τρίτη παράμετρος που δόθηκε κατά την αρχικοποίηση του barrier. Προσέξτε στο σημείο αυτό ότι μετά την αρχικοποίησή του ένας barrier δεν επιτρέπεται να επαναρχικοποιηθεί. Επομένως, θα εξυπηρετεί πάντα ακριβώς N νήματα. Αν χρειαστεί να συγχρονίσουμε ένα διαφορετικό πλήθος νημάτων είτε πρέπει να ορίσουμε έναν νέο barrier είτε μπορούμε να καταστρέψουμε κάποιον υπάρχοντα με την κλήση

```
pthread_barrier_destroy(&bar);
```

Το πρόγραμμα στο Πρόγρ. 4.3, το οποίο υπολογίζει τον μέσο όρο 10 στοιχείων, έχει ένα λογικό σφάλμα. Ενώ η μεταβλητή `avg` θα υπολογιστεί ορθά μιας και έχουμε φροντίσει για τον απαραίτητο αμοιβαίο αποκλεισμό, εντούτοις ενδέχεται να μην *εκτυπωθεί* σωστά! Ο λόγος είναι ότι το νήμα 2 που έχει αναλάβει την εκτύπωση, είναι πολύ πιθανό να φτάσει

```
int                 A[10];
double              avg = 0.0;
pthread_mutex_t     mx = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t bar;           /* Initialized in main() */

void *threadfunc1(void *arg) {
  int i, q1 = 0;

  for (i = 0; i < 5; i++)
    q1 += A[i];
  pthread_mutex_lock(&mx);
  avg = avg + (q1 / 10.0);       /* critical section */
  pthread_mutex_unlock(&mx);
  pthread_barrier_wait(&bar);
  ...
}

void *threadfunc2(void *arg) {
  int i, q2 = 0;

  for (i = 5; i < 10; i++)
    q2 += A[i];
  pthread_mutex_lock(&mx);
  avg = avg + (q2 / 10.0);       /* critical section */
  pthread_mutex_unlock(&mx);
  pthread_barrier_wait(&bar);    /* wait for completion */
  printf("avg = %lf\n", avg);    /* show result */
  ...
}
```

Πρόγραμμα 4.6 *Τελική, ορθή εκδοχή του κώδικα από το Πρόγρ. 4.3.*

στη γραμμή 24, πριν προλάβει το πρώτο νήμα να ολοκληρώσει τη δική του συνεισφορά στο αποτέλεσμα και άρα, να μην εκτυπωθεί η τελική, σωστή τιμή. Εδώ, έχουμε μία κλασική περίπτωση όπου απαιτείται συγχρονισμός μεταξύ των νημάτων. Συγκεκριμένα, κάθε νήμα πρέπει να περιμένει το άλλο να ολοκληρώσει τους υπολογισμούς του, πριν χρησιμοποιήσει το avg παρακάτω. Στο Πρόγρ. 4.6 δίνουμε την τελική και ορθή εκδοχή του κώδικα, όπου γίνεται χρήση ενός barrier για τον απαραίτητο συγχρονισμό. Μετά τον υπολογισμό του, κάθε νήμα εισέρχεται στο barrier (γραμμές 14 και 26). Όταν εισέλθουν και τα δύο, είναι πλέον εγγυημένο ότι έχουν ολοκληρωθεί όλοι οι προηγούμενοι υπολογισμοί και, επομένως, το avg έχει πάρει την τελική του τιμή. Συνεχίζοντας και τα δύο νήματα την εκτέλεσή τους, το νήμα δύο θα εκτυπώσει το τελικό αποτέλεσμα (γραμμή 27).

```
void *threadfunc(void *arg) {
  int id = (int) arg;
  ...
}

int main() {
  int       i;
  pthread_t thrid[5];

  for (i = 0; i < 5; i++)   /* Create threads; give ids 0 .. 4 */
    pthread_create(&thrid[i], NULL, threadfunc, (void *) i);
  ...
}
```

Πρόγραμμα 4.7 *Απόδοση δικής μας ταυτότητας στα νήματα.*

4.6 Αρχικές προγραμματιστικές τεχνικές

Αυτή τη στιγμή γνωρίζουμε τις βασικές δομές που απαιτούνται για τον προγραμματισμό στο μοντέλο κοινόχρηστου χώρου διευθύνσεων και πώς αυτές υλοποιούνται στη βιβλιοθήκη των νημάτων POSIX. Είμαστε λοιπόν, έτοιμοι να σχεδιάσουμε τα πρώτα μας παράλληλα προγράμματα, και στις επόμενες ενότητες θα το κάνουμε μέσα από απλά παραδείγματα εφαρμογών. Πριν, όμως, προχωρήσουμε θα δούμε πώς μπορούμε να έχουμε κάποιες χρήσιμες ευκολίες.

Όπως είδαμε στην Ενότητα 4.3, κάθε νήμα έχει ένα αναγνωριστικό, τύπου `pthread_t`, το οποίο αποδίδεται αυτόματα από το σύστημα κατά τη δημιουργία του νήματος. Επειδή το αναγνωριστικό αυτό, τις περισσότερες φορές, δεν είναι ιδιαίτερα βολικό στη διαμοίραση των υπολογισμών στα νήματα, προτιμούμε να τους δίνουμε εμείς μία δική μας *ταυτότητα*, ώστε να διαχειριζόμαστε N νήματα σαν να έχουν αριθμηθεί ακολουθιακά, 0, 1, 2, ..., $N-1$. Η πιο βολική μέθοδος για να το κάνουμε αυτό είναι να περάσουμε την ταυτότητά τους στο μοναδικό όρισμα της συνάρτησης που θα εκτελέσουν, όπως στο Πρόγρ. 4.7. Προσέξτε ότι, τυπικά, αυτό δεν είναι σωστή προγραμματιστική τεχνική, διότι ο γονέας μετατρέπει έναν ακέραιο σε δείκτη (casting στη γραμμή 11), και στη συνέχεια το νήμα μετατρέπει τον δείκτη σε ακέραιο (casting στη γραμμή 2). Αυτό προϋποθέτει ότι το μέγεθος του ακεραίου και του δείκτη είναι ίδια, κάτι που δεν συμβαίνει σε όλα τα συστήματα[6] και άρα, υπάρχει κίνδυνος να «κοπούν» κάποια bits κατά τις μετατροπές (πολλοί μεταφραστές το επισημαίνουν αυτό με σχετική προειδοποίηση). Παρ' όλα αυτά, ο ακέραιος χρειάζεται σχεδόν πάντα ίσο ή μικρότερο χώρο από έναν δείκτη και επομένως, ο ακέραιος που θα περαστεί στην `pthread_create()` θα καταλήξει με σωστή τιμή στο νήμα. Για τον λόγο αυτό, και εξαιτίας

[6] Για παράδειγμα, ενώ στα συστήματα των 32-bits και οι ακέραιοι και οι δείκτες γενικά καταλαμβάνουν 32 bits, στα συνήθη συστήματα των 64-bit ο ακέραιος καταλαμβάνει 32 bits ενώ ο δείκτης 64 bits.

```c
typedef struct {
  int    id;
  double d;
} thrarg_t;

void *threadfunc(void *arg) {
  int    id = ((thrarg_t *) arg)->id;
  double d  = ((thrarg_t *) arg)->d;
  ...
}

int main() {
  pthread_t thr;
  thrarg_t  data = { 1, 12.5 };
  ...
  pthread_create(&thr, NULL, threadfunc, (void *) &data);
  ...
}
```

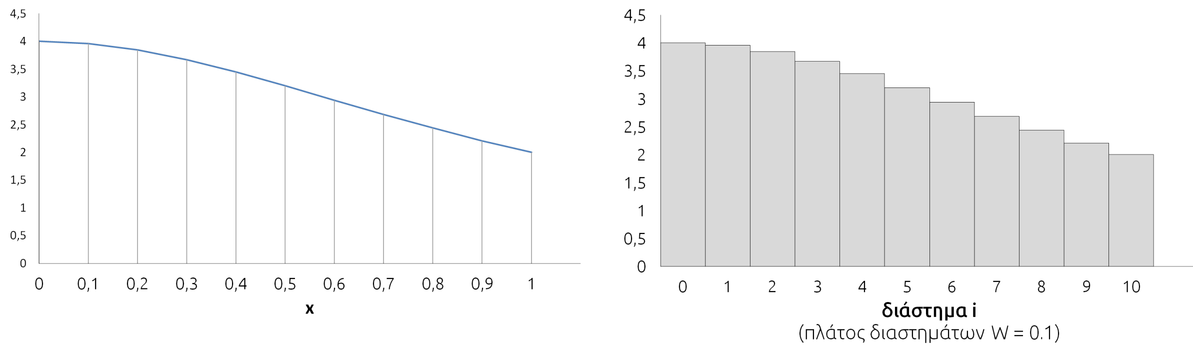
Πρόγραμμα 4.8 *Πέρασμα πολλαπλών δεδομένων σε ένα νήμα.*

της απλότητάς της, η τεχνική χρησιμοποιείται ευρέως.

Ένα δεύτερο σημείο είναι ότι πολλές φορές χρειάζεται να περάσουμε παραπάνω από ένα δεδομένο σε κάποιο νήμα που δημιουργούμε. Δυστυχώς, αυτό δεν είναι άμεσα δυνατό, καθώς η συνάρτηση του νήματος πρέπει υποχρεωτικά να δέχεται μία παράμετρο τύπου δείκτη. Στις περιπτώσεις αυτές, η απλούστερη μέθοδος είναι να περαστεί δείκτης σε μία δομή (struct) η οποία περιέχει τα δεδομένα που θέλουμε να δώσουμε στο νήμα. Ένα μικρό παράδειγμα δίνεται στο Πρόγρ. 4.8 όπου, αφού ορίστηκε ένας νέος τύπος δομής (thrarg_t, γραμμή 4), ο γονέας περνά στο νήμα δύο δεδομένα, την ταυτότητά του και έναν πραγματικό αριθμό διπλής ακρίβειας (γραμμή 16). Το νήμα λαμβάνει τα δεδομένα, με το απαραίτητο casting (γραμμές 7–8).

4.6.1 Υπολογισμός της σταθεράς π = 3.14159...

Στην ενότητα αυτή, σκοπός μας είναι να υπολογίσουμε με ακρίβεια την τιμή της μαθηματικής σταθεράς $\pi = 3.14159265359\ldots$. Είναι γνωστό ότι το ολοκλήρωμα της συνάρτησης $f(x) = 4/(1+x^2)$ από το 0 μέχρι το 1 μας δίνει την επιθυμητή τιμή. Ως γνωστόν, το ολοκλήρωμα είναι ίσο με το εμβαδόν που περικλείεται κάτω από την καμπύλη. Προκειμένου να υπολογίσουμε το εμβαδόν αυτό, θα κάνουμε χρήση αριθμητικής ολοκλήρωσης κατά Riemann, η οποία το προσεγγίζει με ένα άθροισμα εμβαδών από στενά τετράγωνα. Η διαδικασία φαίνεται στο Σχ. 4.9. Χωρίς να μπούμε σε περισσότερες μαθηματικές λεπτομέρειες,



Σχήμα 4.9 *Η συνάρτηση $f(x) = 4/(1+x^2)$ (αριστερά) και η προσέγγιση του εμβαδού της (δεξιά).*

τελικά απαιτείται ο υπολογισμός του παρακάτω αθροίσματος:

$$\pi = \int_0^1 \frac{4}{1+x^2} \approx \sum_{i=0}^{N-1} \frac{4W}{1+[(i+0.5)W]^2},$$

όπου N είναι ο αριθμός των διαστημάτων στα οποία χωρίζουμε την περιοχή $[0,1]$ (όσο πιο πολλά τόσο καλύτερη η προσέγγιση) και W είναι το πλάτος του κάθε διαστήματος, που θα είναι φυσικά, ίσο με $W = 1/N$.

Αν προγραμματίζαμε σειριακά, θα χρησιμοποιούσαμε τον εξής κώδικα, ο οποίος υπολογίζει σε N επαναλήψεις το παραπάνω άθροισμα:

```
#define N 100000
int     i;
double  pi = 0.0, W = 1.0/N;

for (i = 0; i < N; i++)
  pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
```

Μία πρώτη προσέγγιση, όταν κάποιος προγραμματίζει πρώτη φορά παράλληλα, είναι να δημιουργήσει όσο πιο πολλά νήματα μπορεί, με την ελπίδα ότι αυτό θα δώσει και τις καλύτερες επιδόσεις. Δυστυχώς, τις περισσότερες φορές αυτό φέρνει το ανάποδο αποτέλεσμα. Ας προσπαθήσουμε να δημιουργήσουμε τόσα νήματα όσες και οι επαναλήψεις του βρόχου for. Η λογική είναι ότι το καθένα θα αναλάβει μόλις μία επανάληψη, και άρα ελάχιστες πράξεις, και θα τελειώσει πολύ γρήγορα.

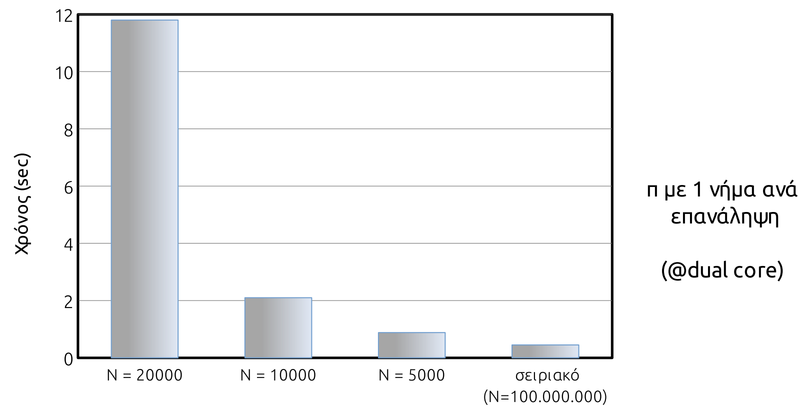
Ο κώδικας δίνεται στο Πρόγρ. 4.10. Το pi είναι καθολική (άρα κοινόχρηστη) μεταβλητή, στην οποία θα συνεισφέρουν όλα τα νήματα. Ο γονέας δημιουργεί N νήματα και στο καθένα περνάει ως όρισμα τον αριθμό της επανάληψης που θα πρέπει να εκτελέσει (γραμμή 18), ενώ στη συνέχεια περιμένει μέχρι αυτά να τερματίσουν (19–20). Κάθε νήμα εκτελεί την επανάληψη που του αντιστοιχεί (γραμμή 9), έχοντας πάρει τα μέτρα του για τον απαιτούμενο αμοιβαίο αποκλεισμό, μιας και επηρεάζεται η κοινόχρηστη μεταβλητή pi.

```
1  #define N 100000
2  double  pi = 0.0, W = 1.0/N;
3  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
4
5  void *pifunc(void *iter) {
6    int i = (int) iter;
7
8    pthread_mutex_lock(&lock);  /* Mutual exclusion */
9    pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
10   pthread_mutex_unlock(&lock);
11 }
12
13 int main() {
14   int i;
15   pthread_t thr[N];          /* Remember the thread ids */
16
17   for (i = 0; i < N; i++)
18     pthread_create(&thr[i], NULL, pifunc, (void *) i);
19   for (i = 0; i < N; i++)
20     pthread_join(thr[i], NULL);
21   printf("pi = %.10lf\n", pi);
22   return 0;
23 }
```

Πρόγραμμα 4.10 *1ος παράλληλος υπολογισμός του π: ένα νήμα ανά επανάληψη. (sas-pi1.c)*

Η ιδέα να δημιουργήσουμε μεγάλο αριθμό νημάτων, όπου στο συγκεκριμένο παράδειγμα φτάνουν και τις εκατοντάδες χιλιάδες δεν είναι πολύ καλή.

- Πρώτα από όλα, τόσα πολλά νήματα πιθανόν να μην επιτρέπεται καν να τα δημιουργήσουμε στα περισσότερα συστήματα.
- Δεύτερον, η δουλειά που κάνει το καθένα από αυτά είναι πολύ μικρή· οδηγούμαστε δηλαδή σε λεπτόκοκκο (fine-grain) παραλληλισμό, τόσο ώστε ο χρόνος που απαιτείται για να δημιουργηθεί το νήμα είναι μάλλον μεγαλύτερος από τον χρόνο που το νήμα χρειάζεται για να κάνει τους υπολογισμούς του.
- Στη (λογική) περίπτωση που ο υπολογιστής που εκτελεί το πρόγραμμα έχει πολύ λιγότερους επεξεργαστές ή πυρήνες, τότε αναγκάζουμε το σύστημα να κάνει χρονομερισμό μεταξύ των νημάτων. Πολλαπλά νήματα θα ανατεθούν σε κάθε πυρήνα και θα υπάρχουν πιθανώς χρονοβόρες εναλλαγές (context switches) μεταξύ τους, επιβαρύνοντας περαιτέρω την εκτέλεση του προγράμματος.
- Ακόμα και τίποτε από τα παραπάνω να μη συνέβαινε, υπάρχει απίστευτος συ-



Σχήμα 4.11 *Χρόνοι εκτέλεσης για τον υπολογισμό του π σε έναν διπύρηνο υπολογιστή, με χρήση ενός διαφορετικού νήματος για κάθε επανάληψη του βρόχου for.*

ναγωνισμός στην κλειδαριά. Τα χιλιάδες νήματα που δημιουργούμε δεν έχουν πολλή δουλειά να κάνουν και σχεδόν ταυτόχρονα συναγωνίζονται να κλειδώσουν την κλειδαριά και να επηρεάσουν την καθολική μεταβλητή. Το αποτέλεσμα είναι μια σχεδόν σειριακή εκτέλεση της γραμμής 9 των νημάτων, επαυξημένη με μεγάλους χρόνους αναμονής στην κλειδαριά.

Το μοιραίο αποτέλεσμα φαίνεται γλαφυρά στο Σχ. 4.11. Όσο λιγότερα νήματα χρησιμοποιούμε, τόσο γρηγορότερα τερματίζει το πρόγραμμα. Φυσικά, ούτε λόγος για την ακρίβεια των υπολογισμών με $N = 5000$ επαναλήψεις / νήματα. Το σειριακό πρόγραμμα και ταχύτερο είναι αλλά και με εξαιρετική ακρίβεια, μιας και χρησιμοποιεί $N = 100.000.000$ διαστήματα.

Το συμπέρασμα από τις παραπάνω παρατηρήσεις είναι ότι, σε γενικές γραμμές, ο μεγάλος αριθμός νημάτων είναι ανεπιθύμητος. Στην πλειονότητα των περιπτώσεων, η βέλτιστη στρατηγική είναι *να χρησιμοποιούμε τόσα νήματα όσοι και οι επεξεργαστικοί πυρήνες που διαθέτουμε*. Με αυτόν τον τρόπο αποφεύγουμε την υπερφόρτωση των επεξεργαστών, όπου αναλαμβάνουν να εκτελέσουν περισσότερη εργασία από αυτή που μπορούν να φέρουν εις πέρας στη μονάδα του χρόνου (με άλλα λόγια, την ανάληψη εκτέλεσης παραπάνω νημάτων από ό,τι μπορούν να εξυπηρετήσουν ταυτόχρονα). Η κατάσταση αυτή είναι γνωστή και ως *υπερεγγραφή* (*oversubscription*) των επεξεργαστών.

Από τη στιγμή που θα χρησιμοποιήσουμε λιγότερα νήματα από το πλήθος των επαναλήψεων του βρόχου for, το βασικό πρόβλημα είναι πώς θα κατανείμουμε τις επαναλήψεις σε αυτά. Η κατανομή των επαναλήψεων στα νήματα είναι γνωστή και ως *δρομολόγηση* (scheduling) του βρόχου. Προφανώς, υπάρχουν πολλοί διαφορετικοί τρόποι να γίνει το μοίρασμα αυτό. Ένας, μάλλον ενστικτώδης τρόπος είναι η λεγόμενη *τμηματική δρομολόγηση* (block scheduling): αν αναλογούν K επαναλήψεις στο κάθε νήμα, τότε οι πρώτες K θα δοθούν στο νήμα 0, οι επόμενες K στο νήμα 1, κ.ο.κ. Επομένως, το νήμα t, $t \geq 0$, θα εκτελέσει

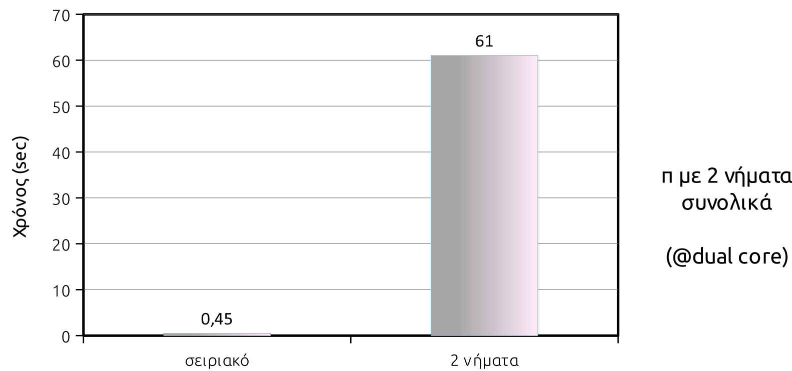
```
1  #define N 10000000       /* # intervals for accuracy */
2  #define NCORES 2         /* # cores in the system */
3  int     WORK;            /* # iterations per thread */
4  double  pi = 0.0, W = 1.0/N;
5  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
6
7  void *pifunc(void *iter) {
8    int i, me = (int) iter;
9
10   for (i = me*WORK; i < (me+1)*WORK; i++) {
11     pthread_mutex_lock(&lock);
12     pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
13     pthread_mutex_unlock(&lock);
14   }
15 }
16
17 int main() {
18   int i;
19   pthread_t thr[NCORES];
20
21   WORK = N / NCORES;             /* work per thread */
22   for (i = 0; i < NCORES; i++)  /* # threads = # cores */
23     pthread_create(&thr[i], NULL, pifunc, (void *) i);
24   for (i = 0; i < NCORES; i++)
25     pthread_join(thr[i], NULL);
26   printf("pi = %.10lf\n", pi);
27   return 0;
28 }
```

Πρόγραμμα 4.12 *2ος παράλληλος υπολογισμός του π: τόσα νήματα όσοι και οι πυρήνες. (sas-pi2.c)*

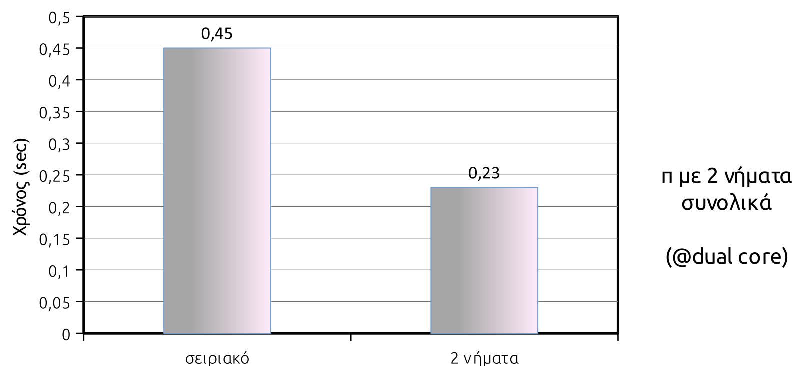
τις επαναλήψεις $tK, tK+1, tK+2, \ldots, tK+K-1 = t(K+1)-1$.

Η λογική αυτή ακολουθείται στο Πρόγρ. 4.12. Το πλήθος των νημάτων είναι ίσο με το πλήθος των πυρήνων (NCORES). Σε κάθε νήμα, επομένως, αντιστοιχούν $N/$NCORES επαναλήψεις (WORK, γραμμή 21). Εδώ έχει γίνει η υπόθεση ότι το πλήθος των επαναλήψεων είναι ακέραιο πολλαπλάσιο του πλήθους των νημάτων. Αν αυτό δεν ισχύει, θα πρέπει να φροντίσουμε το τελευταίο νήμα να εκτελέσει διαφορετικό πλήθος επαναλήψεων από τα άλλα. Οι επιδόσεις του νέου προγράμματος απεικονίζονται στο Σχ. 4.13, και είναι απογοητευτικές. Παρά την αποφυγή του oversubscription, ο χρόνος του παράλληλου προγράμματος είναι εξωφρενικά μεγαλύτερος από αυτόν του σειριακού. Τι πήγε στραβά;

Η απάντηση βρίσκεται στις γραμμές 11–13. Παρά το γεγονός ότι αποφύγαμε τη δημιουργία $N = 10.000.000$ νημάτων, δεν αποφύγαμε τον συναγωνισμό για το κλείδωμα της κλειδαριάς, το οποίο θα γίνει 10.000.000 φορές! Όπως είναι το πρόγραμμα, σε *κάθε* επανά-



Σχήμα 4.13 *Χρόνοι εκτέλεσης του υπολογισμού του π σε έναν διπύρηνο υπολογιστή, με χρήση δύο νημάτων, σύμφωνα με το Πρόγρ. 4.12.*



Σχήμα 4.15 *Χρόνοι εκτέλεσης του υπολογισμού του π σε έναν διπύρηνο υπολογιστή, με χρήση δύο νημάτων και τοπικούς υπολογισμούς, σύμφωνα με το Πρόγρ. 4.14.*

ληψη, απαιτείται η χρήση της κλειδαριάς, ώστε να ενημερωθεί η τιμή του `pi` με ασφάλεια. Ο συναγωνισμός για τις κλειδαριές είναι ίσως *η σημαντικότερη αιτία καθυστερήσεων* στα παράλληλα προγράμματα του μοντέλου κοινόχρηστου χώρου διευθύνσεων. Ως γενικό κανόνα, θα πρέπει να μειώνουμε τη συχνότητα με την οποία απαιτείται η προσφυγή στις κλειδαριές. Πολλές φορές αυτό είναι εύκολο να γίνει, αν απλά υπολογίζουμε ενδιάμεσα αποτελέσματα *τοπικά σε κάθε νήμα*, και στη συνέχεια ενημερώνουμε τα κοινόχρηστα δεδομένα μέσω αμοιβαίου αποκλεισμού.

Εφαρμόζοντας τη στρατηγική αυτή, καταλήγουμε στην τελική έκδοση του υπολογισμού του π, στο Πρόγρ. 4.14. Το κάθε νήμα υπολογίζει το άθροισμα στην ιδιωτική του μεταβλητή `sum` (γραμμές 11–12) και μόνο μία φορά θα καταφύγει στην κλειδαριά για να ενημερώσει το `pi` (γραμμή 14). Όπως δείχνει και το Σχ. 4.15, οι δύο πυρήνες, τελικά, μπορούν να μας προσφέρουν το π στον μισό χρόνο!

Κλείνοντας την ενότητα με το πρώτο μας ολοκληρωμένο πρόγραμμα, σημειώνουμε πάλι ότι η διαμοίραση των επαναλήψεων του βρόχου for έγινε με την υπόθεση ότι το πλήθος των νημάτων διαιρεί ακριβώς το πλήθος των επαναλήψεων. Σε αντίθετη περίπτωση,

```
#define N 10000000       /* # intervals for accuracy */
#define NCORES 2         /* # cores in the system */
int     WORK;            /* # iterations per thread */
double  pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *pifunc(void *iter) {
  int    i, me = (int) iter;
  double sum = 0.0;

  for (i = me*WORK; i < (me+1)*WORK; i++)
    sum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
  pthread_mutex_lock(&lock);
  pi += sum;
  pthread_mutex_unlock(&lock);
}

int main() {
  int i;
  pthread_t thr[NCORES];

  WORK = N / NCORES;             /* work per thread */
  for (i = 0; i < NCORES; i++)  /* # threads = # cores */
    pthread_create(&thr[i], NULL, pifunc, (void *) i);
  for (i = 0; i < NCORES; i++)
    pthread_join(thr[i], NULL);
  printf("pi = %.10lf\n", pi);
  return 0;
}
```

Πρόγραμμα 4.14 *3ος παράλληλος υπολογισμός του π: τοπικοί υπολογισμοί. (saspi3.c)*

πρέπει να ληφθεί μέριμνα για τις επαναλήψεις που περισσεύουν. Ένας άλλος τύπος δρομολόγησης λειτουργεί σε όλες τις περιπτώσεις, χωρίς να χρειάζεται να κάνουμε υποθέσεις για τα πλήθη επαναλήψεων και νημάτων. Πρόκειται για τεχνική που αναφέρεται μερικές φορές και ως διάσπαση βρόχου (loop splitting). Ο όρος είναι μάλλον αδόκιμος, ενώ επιπλέον χρησιμοποιείται αυτούσιος και στο χώρο των μεταφραστών για συγκεκριμένη τεχνική βελτιστοποίησης. Εμείς θα την ονομάζουμε *δρομολόγηση με άλματα* και η ιδέα στην οποία στηρίζεται είναι η εξής: αντί το κάθε νήμα να αναλάβει `WORK` συνεχόμενες επαναλήψεις, αναλαμβάνει επαναλήψεις που απέχουν μεταξύ τους κατά `NPROC`. Αν, για παράδειγμα, έχουμε 5 νήματα, το νήμα 0 θα αναλάβει τις επαναλήψεις 0, 5, 10, ..., το νήμα 1 θα αναλάβει τις επαναλήψεις 1, 6, 11, ..., κ.ο.κ. Αν εφαρμόσουμε τη δρομολόγηση με άλματα, η συνάρτηση που θα εκτελούσε το κάθε νήμα, αντί για αυτή που δίνεται στο Πρόγρ. 4.14, θα ήταν η

παρακάτω:

```
7  void *pifunc(void *iter) {
8    int    i, me = (int) iter;
9    double sum = 0.0;
10
11   for (i = me; i < N; i += NPROC)
12     sum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
13   pthread_mutex_lock(&lock);
14   pi += sum;
15   pthread_mutex_unlock(&lock);
16 }
```

Η μόνη διαφορά είναι στη γραμμή 11. Προσέξτε ότι πλέον δεν χρειάζεται να έχουμε κάποια σχέση μεταξύ N και NPROC. Όλες οι επαναλήψεις θα καλυφτούν, και δεν πρόκειται να ξεφύγουμε πέρα από τη N-οστή. Από την άλλη, θα πρέπει να γνωρίζουμε ότι η δρομολόγηση με άλματα δεν εξασφαλίζει πάντα ισοκατανομή επαναλήψεων (δείτε και το Πρόβλημα 4.5).

4.7 Παραλληλοποίηση πολλαπλών βρόχων

Σε πολλές εφαρμογές το μεγαλύτερο μέρος του χρόνου καταναλώνεται σε επαναληπτικούς βρόχους. Η παραλληλοποίηση είναι επομένως, προσανατολισμένη στο να μοιράσει τις επαναλήψεις των βρόχων αυτών με τέτοιον τρόπο, ώστε όλα τα νήματα να έχουν να εκτελέσουν πάνω-κάτω την ίδια ποσότητα υπολογισμών. Η τμηματική δρομολόγηση και η δρομολόγηση με άλματα είναι δύο πολύ συνηθισμένοι τρόποι να γίνει αυτό. Η δεύτερη θα λέγαμε ότι σε γενικές γραμμές είναι απλούστερη. Η τμηματική δρομολόγηση έχει, όμως, ένα χαρακτηριστικό που την κάνει προτιμητέα σε πολλές περιπτώσεις: οδηγεί σε αυξημένη *τοπικότητα* (locality). Ως παράδειγμα, φανταστείτε ότι κάνετε πολυνηματική πρόσθεση των στοιχείων ενός διανύσματος. Αν χρησιμοποιήσετε δρομολόγηση με άλματα, τότε ένα νήμα θα προσθέσει στοιχεία διάσπαρτα στη μνήμη, μη συνεχόμενα. Κάτι τέτοιο όμως, οδηγεί σε μεγάλες καθυστερήσεις σε όλα τα σύγχρονα συστήματα με κρυφές μνήμες (cache). Η κρυφή μνήμη στηρίζεται στην ύπαρξη τοπικότητας των υπολογισμών για να παρέχει αυξημένες ταχύτητες. Μαζί με το στοιχείο που ζητά, ο επεξεργαστής φέρνει και τα γειτονικά του στοιχεία, προκαταβολικά, με την ελπίδα ότι θα φανούν χρήσιμα. Αν αυτά τα γειτονικά στοιχεία τελικά δεν τα χρησιμοποιήσει το νήμα, όπως στην περίπτωση της δρομολόγησης με άλματα, τότε η κρυφή μνήμη λειτουργεί εις βάρος του συστήματος. Με την τμηματική δρομολόγηση, όμως, η κρυφή μνήμη όχι μόνο δεν επιβαρύνει, αλλά αντίθετα επιταχύνει στο μέγιστο βαθμό, αφού κάθε νήμα προσθέτει συνεχόμενα (και άρα γειτονικά στη μνήμη) στοιχεία.

Εκτός από τις δύο αυτές δρομολογήσεις, έχουν χρησιμοποιηθεί και αρκετές άλλες με θετικά και αρνητικά στοιχεία η κάθε μία. Μερικές θα τις δούμε παρακάτω, όταν θα

μιλήσουμε για το OpenMP. Πολλές φορές όμως, οι επαναλήψεις ενός βρόχου είναι σχετικά λίγες για να κρατήσουν όλα τα νήματα απασχολημένα. Η κατάσταση βελτιώνεται αν μέσα στο βρόχο υπάρχει εμφωλευμένος και δεύτερος βρόχος, οπότε υπάρχουν περισσότερες δυνατότητες για παραλληλοποίηση.

4.7.1 Πίνακας επί διάνυσμα

Θέλουμε να προγραμματίσουμε τον υπολογισμό του γινομένου ενός πίνακα επί ένα διάνυσμα. Εάν το διάνυσμα `v` έχει N στοιχεία και ο πίνακας `A` είναι διάστασης $N \times N$, το γινόμενό τους (διάνυσμα `res`) θα μπορούσε να υπολογιστεί σειριακά ως εξής:

```
for (i = 0; i < N; i++) {
  for (sum = j = 0; j < N; j++)
    sum += A[i][j]*v[j];
  res[i] = sum;
}
```

Για να παραλληλοποιήσουμε το πρόγραμμα με πολλαπλά νήματα, θα μπορούσαμε να ακολουθήσουμε οποιαδήποτε από τις τεχνικές που έχουμε μάθει μέχρι τώρα, προκειμένου να δρομολογήσουμε το βρόχο του `i`. Για παράδειγμα, θα μπορούσαμε να χρησιμοποιήσουμε τμηματική δρομολόγηση και να μοιράσουμε τις N επαναλήψεις του σε `NPROC` νήματα[7]. Τι γίνεται όμως αν ο παράλληλος υπολογιστής που διαθέτουμε έχει μεγάλο αριθμό πυρήνων σε σχέση με τη διάσταση του πίνακα (N);

Έστω ότι ο παράλληλος υπολογιστής μας διαθέτει `NPROC` = 32 πυρήνες. Έστω, επίσης, ότι ο πίνακας που θέλουμε να πολλαπλασιάσουμε έχει 256 στοιχεία—είναι δηλαδή διάστασης 16×16 ($N = 16$). Αν δημιουργήσουμε 32 νήματα, μόνο 16 από αυτά θα κάνουν κάτι, και άρα οι 16 από τους 32 πυρήνες θα είναι μονίμως ανενεργοί κατά τη διάρκεια εκτέλεσης. Είναι φανερό ότι σε αυτή την περίπτωση η απλή παραλληλοποίηση που κάναμε δεν είναι καθόλου καλά σχεδιασμένη.

Η λύση είναι, αντί να παραλληλοποιήσουμε την ακολουθία των γραμμών του πίνακα (δηλαδή τον βρόχο `i`), να μοιράσουμε τους υπολογισμούς *όλων* των στοιχείων του πίνακα: *παραλληλοποίηση και των δύο βρόχων μαζί* (`i` και `j`). Επομένως, αυτό που θα προσπαθήσουμε να κάνουμε είναι μία παραλληλοποίηση με τέτοιο τρόπο, ώστε κάθε νήμα να εκτελεί ορισμένες από τις επαναλήψεις του βρόχου `i` αλλά και ορισμένες από τις επαναλήψεις του βρόχου `j`.

Αν υποθέσουμε ότι `NPROC` = 20 και $N = 10$, τότε κάθε επανάληψη του βρόχου `i` θα πρέπει να την αναλάβουν, πλέον, `NPROC`$/N$, δηλαδή δύο νήματα—το κάθε ένα θα υπολογίσει 5 από τις επαναλήψεις του βρόχου `j`. Έτσι, τα νήματα 0 και 1 θα πρέπει να αναλάβουν το `i` = 0, τα νήματα 2 και 3 θα πρέπει να αναλάβουν το `i` = 1 κλπ. Στη συνέχεια, τα νήματα 0 και 1 θα αναλάβουν από κοινού την εκτέλεση του βρόχου `j` για `i` = 0. Το νήμα 0 μπορεί

[7] Δείτε τα Προβλήματα 4.6 και 4.7.

```
1  #define N     100       /* vector size */
2  #define NPROC 200       /* #cores = #threads */
3  #define TPR   NPROC/N   /* threads-per-row */
4  double  A[N][N], v[N], res[N];
5  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
6
7  void *matvec(void *arg) {
8    int    i, j, me = (int) arg;
9    double sum = 0.0;
10
11   i = me / TPR;        /* my row */
12   for (j = me % TPR; j < N; j += TPR)
13     sum += A[i][j]*v[j];
14   pthread_mutex_lock(&lock);
15   res[i] += sum;
16   pthread_mutex_unlock(&lock);
17 }
18
19 int main() {
20   int i;
21   pthread_t thr[NPROC];
22
23   for (i = 0; i < N; i++)
24     res[i] = 0.0;
25   for (i = 0; i < NPROC; i++)
26     pthread_create(&thr[i], NULL, matvec, (void *) i);
27   for (i = 0; i < NPROC; i++)
28     pthread_join(thr[i], NULL);
29   show_result(res, N);
30   return 0;
31 }
```

Πρόγραμμα 4.16 *Παραλληλοποίηση δύο βρόχων (υποθέτουμε ότι το N διαιρεί ακριβώς το NPROC). (sas-mxv1.c)*

να εκτελέσει τις επαναλήψεις για j = 0, 1, ..., 4 και το νήμα 1 τις υπόλοιπες πέντε (j = 5, 6, ..., 9). Γενικά, NPROC$/N$ νήματα αναλαμβάνουν το i = 0, άλλα τόσα αναλαμβάνουν το i = 1, κ.ο.κ. Τα NPROC$/N$ νήματα που αναλαμβάνουν την ίδια επανάληψη του i θα πρέπει να μοιραστούν τις N επαναλήψεις του j. Η διαμοίραση των επαναλήψεων τόσο του i όσο και του j μπορεί να γίνει είτε με τμηματική δρομολόγηση είτε με δρομολόγηση με άλματα.

Στο Πρόγρ. 4.16 έχουν παραλληλοποιηθεί και οι δύο βρόχοι (i και j) του πολλαπλασιασμού πίνακα επί διανύσμα, με βάση τη συζήτηση που προηγήθηκε. Για τη μελέτη του κώδικα αυτού, θα πρέπει να έχει ασχοληθεί κανείς με τα Προβλήματα 4.6 και 4.7. Σύμφωνα με αυτά που είπαμε, TPR $=$ NPROC$/N$ νήματα αναλαμβάνουν την ίδια επανάληψη

του εξωτερικού βρόχου i. Συγκεκριμένα, τα πρώτα TPR νήματα αναλαμβάνουν τη γραμμή 0 του πίνακα, τα επόμενα TPR τη γραμμή 1, κ.ο.κ. (εντολή στη γραμμή 11 του κώδικα). Οι επαναλήψεις του βρόχου j μοιράζονται μεταξύ των αντίστοιχων TPR νημάτων με δρομολόγηση με άλματα (γραμμές 12–13 στον κώδικα). Προσέξτε, όμως, τώρα ότι την κοινόχρηστη μεταβλητή res[i] την επηρεάζουν *και τα* TPR νήματα, που αναλαμβάνουν την ίδια επανάληψη του i. Επομένως, είναι απαραίτητος ο αμοιβαίος αποκλεισμός (γραμμές 14–16). Εδώ, βέβαια, χρησιμοποιούμε μόνο μία κλειδαριά για όλα τα νήματα. Θα ήταν αποδοτικότερο να είχαμε μία διαφορετική κλειδαριά ανά TPR νήματα, ώστε να αποφεύγουμε τον μεγάλο συναγωνισμό.

Εκτός αυτού, είναι απαραίτητη πλέον και η αρχικοποίηση του (κάθε) res[i] στο μηδέν, αφού τα δύο νήματα που το επηρεάζουν προσθέτουν κάτι σε αυτό. Έτσι δικαιολογούνται οι γραμμές 23–24 στον κώδικα. Η αρχικοποίηση αυτή εισάγει, όμως, μία καθυστέρηση της τάξης του $O(N)$, η οποία μπορεί να μην είναι καθόλου αμελητέα. Για τον λόγο αυτό, είναι πάντα καλύτερο την αρχικοποίηση κάθε στοιχείου να την αναλαμβάνει διαφορετικό νήμα, ώστε η αρχικοποίηση όλου του res να γίνει παράλληλα. Συγκεκριμένα, θα μπορούσε από τα TPR νήματα που αναλαμβάνουν το ίδιο i, αυτό με τη μικρότερη ταυτότητα να κάνει την αρχικοποίηση του res[i] στο μηδέν, πριν αρχίσει τους υπόλοιπους υπολογισμούς. Τότε όμως, υπάρχει ένα άλλο πρόβλημα: πώς θα είμαστε σίγουροι ότι το νήμα αυτό θα προλάβει να αρχικοποιήσει το res[i], πριν ξεκινήσουν υπολογισμούς τα υπόλοιπα; Ο μόνος τρόπος να σιγουρευτούμε είναι η χρήση barrier. Και αυτό είναι που γίνεται στο Πρόγρ. 4.17. Όπως και με την κλειδαριά, το καλύτερο θα ήταν να έχουμε έναν barrier ανά TPR νήματα, και όχι έναν συνολικά επάνω στον οποίον συγχρονίζονται όλα τα νήματα της εφαρμογής.

4.7.2 Πίνακας επί πίνακα

Στην ενότητα αυτή θα δούμε μία ακόμα ενδιαφέρουσα εφαρμογή, αυτή του πολλαπλασιασμού τετραγωνικών πινάκων. Έστω δύο πίνακες A και B, διάστασης $N \times N$. Το γινόμενό τους, C, θα μπορούσε να υπολογιστεί σειριακά ως εξής:

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
  {
    for (k = sum = 0; k < N; k++)
      sum += A[i][k]*B[k][j];
    C[i][j] = sum;
  }
```

Έχουμε δει ήδη τεχνικές για την παραλληλοποίηση του προγράμματος αυτού, ώστε να δημιουργήσουμε ένα αποδοτικό πρόγραμμα κοινόχρηστου χώρου διευθύνσεων. Θα μπορούσαμε για παράδειγμα:

- Να παραλληλοποιήσουμε τον βρόχο του i χρησιμοποιώντας τμηματική δρομολό-

```
#define N     100       /* vector size */
#define NPROC 200       /* #cores = #threads */
#define TPR   NPROC/N   /* threads-per-row */
double  A[N][N], v[N], res[N];
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t bar;

void *matvec(void *arg) {
  int    i, j, me = (int) arg;
  double sum = 0.0;

  i = me / TPR;        /* my row */
  if (me % TPR == 0)     /* i will initialize */
    res[i] = 0;
  pthread_barrier_wait(&bar);   /* make sure all wait here */
  for (j = me % TPR; j < N; j += TPR)
    sum += A[i][j]*v[j];
  pthread_mutex_lock(&lock);
  res[i] += sum;
  pthread_mutex_unlock(&lock);
}

int main() {
  int i;
  pthread_t thr[NPROC];

  pthread_barrier_init(&bar, NULL, NPROC);
  for (i = 0; i < NPROC; i++)
    pthread_create(&thr[i], NULL, matvec, (void *) i);
  for (i = 0; i < NPROC; i++)
    pthread_join(thr[i], NULL);
  show_result(res, N);
  return 0;
}
```

Πρόγραμμα 4.17 *Δεύτερη εκδοχή του Προγρ. 4.16, με παράλληλη αρχικοποίηση του αποτελέσματος και συγχρονισμό. (sas-mxv2.c)*

γηση ή δρομολόγηση με άλματα, αν ο αριθμός των πυρήνων είναι μικρότερος του N. Αυτό ακριβώς ζητείται στο Πρόβλημα 4.8.

- Να παραλληλοποιήσουμε τους βρόχους του `i` και `j` ταυτόχρονα, ιδιαίτερα αν ο αριθμός των επεξεργαστών είναι μεγαλύτερος του N.

Πρακτικά, στην πρώτη μέθοδο, κάθε νήμα θα αναλάβει τον υπολογισμό κάποιων *γραμμών* του τελικού αποτελέσματος. Στη δεύτερη μέθοδο, την οποία θα ακολουθήσουμε

εδώ, αν χρησιμοποιηθεί τμηματική δρομολόγηση για την παραλληλοποίηση και των δύο βρόχων, κάθε νήμα αναλαμβάνει τον υπολογισμκό ενός *υποπίνακα* του αποτελέσματος. Η μέθοδος αυτή ονομάζεται *διαχωρισμός σκακιέρας* (checkerboard partitioning), καθώς το αποτέλεσμα ουσιαστικά χωρίζεται σε τετράγωνα τμήματα (υποπίνακες) τα οποία ανατίθενται στα νήματα. Μαθηματικά, ένας πίνακας $N \times N$ μπορεί να εκφραστεί ως ένας $M \times M$ πίνακας *υποπινάκων* μεγέθους $(N/M) \times (N/M)$ ο καθένας. Για παράδειγμα, ο πίνακας C μπορεί να γραφεί ως εξής:

$$C = \begin{bmatrix} C_{00} & C_{01} & \cdots & C_{0,M-1} \\ C_{10} & C_{11} & \cdots & C_{1,M-1} \\ \vdots & \vdots & & \vdots \\ C_{M-1,0} & C_{M-1,1} & \cdots & C_{M-1,M-1} \end{bmatrix}.$$

Εδώ, π.χ. ο υποπίνακας C_{00} είναι το εξής τμήμα του πίνακα C:

$$C_{00} = \begin{bmatrix} c_{00} & c_{01} & \cdots & c_{0,S-1} \\ c_{10} & c_{11} & \cdots & c_{1,S-1} \\ \vdots & \vdots & & \vdots \\ c_{S-1,0} & c_{S-1,1} & \cdots & c_{S-1,S-1} \end{bmatrix}$$

όπου $S = N/M$. Με την μέθοδο του διαχωρισμού σκακιέρας κάθε νήμα ουσιαστικά υπολογίζει έναν υποπίνακα C_{xy} του πίνακα $C = A \times B$.

Η λύση δίνεται στο Πρόγρ. 4.18. Εδώ, έχουμε υποθέσει ότι κάθε υποπίνακα του αποτελέσματος τον υπολογίζει διαφορετικό νήμα, και επομένως το πλήθος των υποπινάκων είναι ίσο με το πλήθος των νημάτων (NTHR $= M^2$). Στη γενική περίπτωση θα μπορούσε το ίδιο νήμα να αναλάβει να υπολογίζει πολλούς υποπίνακες (βλ. Πρόβλημα 4.9). Στις γραμμές 12–13 το νήμα βρίσκει ποιος υποπίνακας C_{xy} τού αντιστοιχεί και στη συνέχεια υπολογίζει τα στοιχεία του (γραμμές 14–20). Προσέξτε ότι μιας και τα νήματα υπολογίζουν διαφορετικά στοιχεία μεταξύ τους, δεν υπάρχει ανάγκη αμοιβαίου αποκλεισμού. Πρόκειται για μία εντελώς παραλληλοποιήσιμη εφαρμογή.

4.8 Η τεχνική της αυτοδρομολόγησης

Μέχρι τώρα, είδαμε τρόπους να μοιράζουμε τις επαναλήψεις των βρόχων, έτσι ώστε τα νήματα να αναλαμβάνουν ισόποση δουλειά. Εφόσον έχουμε ένα νήμα ανά επεξεργαστικό πυρήνα, η συνολική δουλειά που πρέπει να γίνει μοιράζεται εξίσου σε όλους τους πυρήνες και, επομένως, πετυχαίνουμε την επιθυμητή ισοκατανομή του φόρτου. Τι γίνεται όμως, αν οι επεξεργαστές απασχολούνται ταυτόχρονα και με άλλες εργασίες ή αν δεν έχουν για κάποιο λόγο την ίδια ταχύτητα πάντα; Εφόσον όλα αυτά γίνονται *δυναμικά*, δηλαδή

```
1  #define N    1000     /* matrices 1000x1000 */
2  #define NTHR 25       /* # threads */
3  #define M    5        /* M*M submatrices in total */
4  #define S    N/M      /* size of each submatrix */
5  double  A[N][N], B[N][N], C[N][N];
6
7  void *checker(void *arg) {
8    int    i, j, k, x, y, me = (int) arg;
9    double sum;
10
11   x = me / M;
12   y = me % M;
13   for (i = x*S; i < (x+1)*S; i++) {  /* calculate Cxy  */
14     for (j = y*S; j < (y+1)*S; j++) {
15       for (k = 0, sum = 0.0; k < N; k++)
16         sum += A[i][k]*B[k][j];
17       C[i][j] = sum;
18     }
19   }
20 }
21
22 int main() {
23   int i;
24   pthread_t thr[NTHR];
25
26   for (i = 0; i < NTHR; i++)
27     pthread_create(&thr[i], NULL, checker, (void *) i);
28   for (i = 0; i < NTHR; i++)
29     pthread_join(thr[i], NULL);
30   show_result(C, N);
31   return 0;
32 }
```

Πρόγραμμα 4.18 *Πολλαπλασιασμός τετραγωνικών πινάκων με διαχωρισμό σκακιέρας. (sas-mxm-checker.c)*

αλλάζουν απρόβλεπτα κατά το χρόνο εκτέλεσης της εφαρμογής μας, η ισόποση διαμοίραση των εργασιών είναι πολύ δύσκολο να οδηγήσει στην ταχύτερη δυνατή εκτέλεση.

Η *αυτοδρομολόγηση* (self-scheduling) είναι μία γενική τεχνική, η οποία προσαρμόζει την κατανομή των υπολογισμών αυτόματα κατά τον χρόνο εκτέλεσης και ανάλογα με το φόρτο του κάθε επεξεργαστή. Επομένως επιτυγχάνει, σε μεγάλο βαθμό, μια πιο δίκαια κατανομή των εργασιών. Σε αντίθεση με ό,τι είδαμε μέχρι τώρα, η διαμοίραση των υπολογισμών γίνεται αυτόματα και δυναμικά (δηλαδή κατά την εκτέλεση) από τους ίδιους τους πυρήνες, αντί να είναι προκαθορισμένη από τον προγραμματιστή. Κατά βάση, πρόκειται για μία μορφή της επαναληπτικής διάσπασης που συζητήσαμε στην Ενότητα 1.5.1. Η βασική

ιδέα συνοψίζεται στις παρακάτω 3 γραμμές ψευδοκώδικα:

```
while (there-are-things-to-calculate) {
    Task = get-the-next-task();
    execute(Task);
}
```

Συγκεκριμένα, χωρίζουμε τους υπολογισμούς μας σε μικρά τμήματα τα οποία ονομάζουμε εργασίες (tasks) και αφήνουμε κάθε νήμα να ζητά κάποιο τμήμα για να εργαστεί. Όταν ένα νήμα ολοκληρώσει το έργο του (πάντα ανάλογα με την ταχύτητα του πυρήνα που το εκτελεί), μπορεί να ζητήσει ένα επιπλέον τμήμα για να υπολογίσει κ.ο.κ. Εφόσον έναν πυρήνας έχει μικρό φόρτο ή είναι πολύ γρήγορος, θα προλάβει και θα εκτελέσει πολλές εργασίες, σε αντίθεση με κάποιον που απασχολείται και με άλλα πράγματα ή είναι γενικά πιο αργός. Με αυτόν τον τρόπο επιτυγχάνεται αυτόματα η καλύτερη δυνατή κατανομή του φόρτου, κάτι που μπορεί να οδηγήσει στις καλύτερες δυνατές επιδόσεις.

Στη γενική περίπτωση, οι εργασίες οργανώνονται σε μία ουρά από όπου τα νήματα τις αφαιρούν μία-μία και τις εκτελούν, μέχρι να αδειάσει η ουρά, ενώ πιθανώς δημιουργούνται και εισάγονται νέες κατά τη διάρκεια της εκτέλεσης. Αυτός ο τύπος παράλληλης εκτέλεσης είναι γνωστός και ως *σακί με εργασίες* (bag-of-tasks). Στην περίπτωσή μας, θα ασχοληθούμε με μια πιο απλοϊκή, αλλά πολύ πρακτική και συνήθη μορφή της. Θα υποθέσουμε ότι οι εργασίες είναι καθορισμένες από την αρχή της εκτέλεσης και ότι είναι αριθμημένες από 0 έως `NTASK` − 1. Η συνάρτηση που εκτελεί το κάθε νήμα δίνεται στο Πρόγρ. 4.19. Η κοινόχρηστη μεταβλητή `taskid` δείχνει ποια εργασία είναι η επόμενη που πρέπει να δοθεί προς εκτέλεση. Κάθε νήμα ζητά την τιμή της, και μόλις τη λάβει, μέσω αμοιβαίου αποκλεισμού, την αυξάνει κατά 1 (γραμμές 8–10). Στη συνέχεια εκτελεί την αντίστοιχη εργασία, καλώντας την `taskexecute()` (γραμμή 13). Αν δεν υπάρχουν άλλες εργασίες, το νήμα τερματίζει (γραμμές 11–12).

Ο κώδικας είναι πολύ απλός αλλά και εξαιρετικά αποτελεσματικός.Τα νήματα συναγωνίζονται για την εκτέλεση των στοιχειωδών εργασιών. Όποιος επεξεργαστής έχει μικρό φόρτο, θα προλάβει να εκτελέσει πολλές εργασίες, σε αντίθεση με κάποιον που έχει μεγαλύτερο φόρτο. Με αυτόν τον τρόπο επιτυγχάνεται αυτόματα η καλύτερη δυνατή κατανομή φόρτου. Επιπλέον, ο κώδικας που εκτελεί το κάθε νήμα είναι αυτός που φαίνεται στο Πρόγρ. 4.19, *ανεξαρτήτου εφαρμογής και ανεξαρτήτου πλήθους νημάτων*. Το μόνο που αλλάζει από εφαρμογή σε εφαρμογή είναι η τιμή του `NTASK` και η συνάρτηση `taskexecute()`.

Ας πάρουμε για παράδειγμα, τον υπολογισμό του π που είδαμε στην Ενότητα 4.6.1. Έχουμε διάφορες επιλογές στο ποιες ακριβώς θα είναι οι στοιχειώδεις εργασίες που θα εκτελέσουν τα νήματα. Θα μπορούσαμε, παραδείγματος χάριν, να θεωρήσουμε, ως στοιχειώδη εργασία, τον υπολογισμό μίας επανάληψης του βρόχου `i`. Μία τέτοια επιλογή, όμως, οδηγεί σε πολύ λεπτό κόκκο παραλληλισμού. Θα πρέπει να είναι φανερό ότι μάλλον περισσότερος χρόνος απαιτείται για την απόκτηση της εργασίας (συναγωνισμός στην κλειδαριά) παρά για την εκτέλεσή της. Επομένως, κρίνεται αποτελεσματικότερο να χρησιμοποιήσουμε

```
1  int taskid = 0;      /* the next task id to execute */
2  pthread_mutex_t tlock = PTHREAD_MUTEX_INITIALIZER;
3
4  void *thrfunc(void *arg) {
5    int t;
6
7    while (1) {        /* forever */
8      pthread_mutex_lock(&tlock);
9      t = taskid++;    /* get next task */
10     pthread_mutex_unlock(&tlock);
11     if (t >= NTASK)
12       break;         /* all tasks done */
13     taskexecute(t);
14   }
15 }
```

Πρόγραμμα 4.19 *Σκελετός προγράμματος με αυτοδρομολόγηση.*

πιο χονδρόκοκκες εργασίες. Θα μπορούσαμε, π.χ. να υπολογίζουμε K συνεχόμενους όρους του αθροίσματος σε κάθε εργασία. Έτσι, η t-οστή εργασία θα υπολόγιζε το:

```
for (i = t*K; i < (t+1)*K; i++)
  pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
```

Ο κώδικας της `taskexecute()` δίνεται στο Πρόγρ. 4.20. Προσέξτε ότι, από τη στιγμή που εμπλέκεται η κοινόχρηστη μεταβλητή `pi`, θα απαιτηθεί αμοιβαίος αποκλεισμός για την τροποποίησή της από τις εργασίες. Για τον λόγο αυτό γίνεται χρήση μίας επιπλέον κλειδαριάς `pilock` (γραμμές 13–15), εκτός της `tlock`.

Ένα άλλο παράδειγμα είναι ο υπολογισμός του γινομένου δύο πινάκων που είδαμε στην Ενότητα 4.7.2. Μία στοιχειώδης εργασία θα μπορούσε να ήταν ο υπολογισμός ενός υποπίνακα. Το μέγεθος των υποπινάκων ή το πλήθος τους δεν θα έχει πλέον κάποια σχέση με το πλήθος των νημάτων και άρα, μπορούμε να τα καθορίσουμε κατά το δοκούν. Ο αναγνώστης προτρέπεται να ασχοληθεί με το Πρόβλημα 4.10.

Η αυτοδρομολόγηση είναι μία ισχυρή τεχνική η οποία παρέχει δυναμική εξισορρόπηση φόρτου τόσο σε περιβάλλοντα όπου οι σχετικές ταχύτητες των πυρήνων είναι ανομοιόμορφες, όσο και σε εφαρμογές όπου η κάθε στοιχειώδης εργασία έχει διαφορετικές υπολογιστικές απαιτήσεις. Από την άλλη, θα πρέπει να γίνει κατανοητό ότι εισάγει τις δικές της απαιτήσεις σε χρόνο. Για παράδειγμα, όπως φαίνεται και στο Προγρ. 4.19, τα νήματα υφίστανται καθυστερήσεις λόγω συναγωνισμού για τη λήψη της επόμενης εργασίας. Στη γενικότερη περίπτωση όπου οι εργασίες φυλάσσονται σε ουρά, υπάρχει επιπλέον και το κόστος διαχείρισης της ουράς αυτής. Τέλος, η χρήση αυτοδρομολόγησης απαιτεί την αναδιοργάνωση του κώδικα, ώστε να μπορεί να εκφραστεί ως σύνολο εργασιών. Κάτι τέτοιο, όμως, δεν είναι πάντα εύκολο και σε μερικές περιπτώσεις είναι έως και αδύνατο.

```
1  #define N 10000000      /* # iterations */
2  #define K 100           /* iterations per task */
3  #define NTASK N/K       /* total # tasks */
4  double pi = 0.0, W = 1.0/N;
5  pthread_mutex_t pilock = PTHREAD_MUTEX_INITIALIZER;
6
7  void taskexecute(int t) {
8    int    i;
9    double mysum = 0.0;
10
11   for (i = t*K; i < (t+1)*K; i++)
12     mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
13   pthread_mutex_lock(&pilock);
14   pi += mysum;
15   pthread_mutex_unlock(&pilock);
16 }
```

Πρόγραμμα 4.20 *Εργασίες για τον υπολογισμό του π με αυτοδρομολόγηση. (sas-ss-pi.c)*

4.9 Προγραμματισμός με το OpenMP

Τα νήματα POSIX αποτελούν ίσως τον βασικότερο τρόπο να προγραμματίσει κανείς στο μοντέλο του κοινόχρηστου χώρου διευθύνσεων. Αποτελούν διεθνές πρότυπο και υποστηρίζονται από τα περισσότερα συστήματα. Επίσης, επιτρέπουν λεπτομερή διαχείριση της παράλληλης εκτέλεσης και παρέχουν πολλά προχωρημένα χαρακτηριστικά που δεν τα είδαμε εδώ. Παρ' όλα αυτά, είναι αρκετά επίπονα στη χρήση τους, απαιτούν αρκετή προγραμματιστική εμπειρία, και ως προγραμματιστικές δομές θεωρούνται σχετικά «χαμηλού» επιπέδου, αφού ο προγραμματιστής πρέπει να κανονίσει μόνος του, πλήρως, όλες τις μικρές λεπτομέρειες της δημιουργίας, της εκτέλεσης, της δρομολόγησης και του τερματισμού τους. Επιπλέον, εάν ο σειριακός κώδικας υπάρχει ήδη, απαιτούν επανεγγραφή τμημάτων του και γενική αναδιοργάνωσή του, ώστε να γίνει κατάλληλος για πολυνηματική εκτέλεση. Όλα αυτά τα στοιχεία αποτελούν εμπόδια για την αύξηση της παραγωγικότητας των προγραμματιστών και για τη γενικευμένη χρήση των νημάτων. Τα προβλήματα αυτά ήρθε να λύσει το OpenMP.

Το OpenMP (Open Multi-Processing) αποτελεί ένα ανοιχτό προγραμματιστικό πρότυπο το οποίο αποτελείται από ένα σύνολο γλωσσικών επεκτάσεων για τον καθορισμό του παραλληλισμού, και από ένα σύνολο κλήσεων βιβλιοθήκης για την υποστήριξη της εκτέλεσης των προγραμμάτων. Το OpenMP είναι διαθέσιμο για τις γλώσσες Fortran, C και C++, και σχεδόν όλοι οι σύγχρονοι μεταφραστές το υποστηρίζουν πλήρως. Μπορούμε με ασφάλεια να πούμε ότι πλέον αποτελεί, με διαφορά, τον πιο δημοφιλή τρόπο να προγραμματίζει κανείς στο μοντέλο κοινόχρηστης μνήμης, και οι λόγοι θα γίνουν εύκολα κατανοητοί

παρακάτω.

Ένα βασικό ζητούμενο στην ανάπτυξη του OpenMP ήταν η διατήρηση υπάρχοντος σειριακού κώδικα στην αρχική του μορφή, ώστε να μπορεί χωρίς αλλαγές να μεταφραστεί και να λειτουργήσει όπως είχε σχεδιαστεί. Αυτό γίνεται δυνατό με τη χρήση *οδηγιών* (directives), οι οποίες αγνοούνται από τους κλασικούς σειριακούς μεταφραστές, αλλά χρησιμοποιούνται από μεταφραστές OpenMP για να παραλληλοποιήσουν το πρόγραμμα. Στη Fortran οι οδηγίες εισάγονται μέσα σε σχόλια, ενώ στην C/C++ μέσα σε γραμμές `#pragma` που απευθύνονται στον μεταφραστή. Η γενική μορφή μίας οδηγίας στο OpenMP είναι της μορφής:

```
#pragma omp construct clause clause clause ... <newline>
```

όπου construct είναι η λειτουργία που πρέπει να γίνει στην περιοχή του κώδικα που ακολουθεί, και clauses είναι φράσεις που παραμετροποιούν τη λειτουργία (δεν επιτρέπεται να υπάρχει τίποτε άλλο στην ίδια γραμμή). Ξεκινώντας από ένα υπάρχον σειριακό πρόγραμμα, ο προγραμματιστής μπορεί σταδιακά να προσθέτει οδηγίες OpenMP και να παραλληλοποιεί χωρίς κόπο τμήματα του κώδικα.

Η μετάφραση ενός προγράμματος OpenMP γίνεται με το πέρασμα κατάλληλων ορισμάτων στον μεταφραστή του συστήματος. Για να κάνει τη μετάφραση ο γνωστός GCC της GNU, απαιτείται το όρισμα `-fopenmp`, αλλιώς αγνοούνται όλες οι οδηγίες `#pragma` του OpenMP:

```
% gcc -fopenmp program.c
```

Αγνοώντας τις οδηγίες OpenMP, προκύπτει ένα συντακτικά ορθό σειριακό πρόγραμμα που μπορεί να μεταφραστεί με οποιονδήποτε σειριακό μεταφραστή.

4.9.1 Νήματα και μεταβλητές

Η εκτέλεση ενός προγράμματος που έχει γραφτεί σε OpenMP στηρίζεται στη στρατηγική fork-join, που είδαμε στην αρχή του κεφαλαίου. Συγκεκριμένα, όπου υπάρχει η κατάλληλη οδηγία για παραλληλοποίηση (`#pragma omp parallel`), το νήμα που τη συναντά (*νήμα-γονέας*) δημιουργεί *ομάδα* νημάτων. Η ομάδα, στην οποία συμμετέχει και το νήμα-γονέας ως *αρχηγός* (master), εκτελεί τον κώδικα της παράλληλης περιοχής. Στη συνέχεια, όλα τα νήματα καταστρέφονται πλην του αρχηγού, το οποίο συνεχίζει την εκτέλεση του υπόλοιπου κώδικα.

Η εκτέλεση είναι πολυνηματική, αλλά τα νήματα δεν αποτελούν οντότητες που μπορεί να χειριστεί άμεσα ο προγραμματιστής. Δημιουργούνται και καταστρέφονται αυτόματα στις παράλληλες περιοχές. Έχουν όμως ταυτότητα, και μάλιστα, μέσα σε μία ομάδα N νημάτων, οι ταυτότητες είναι ακολουθιακά αριθμημένες από 0 (για τον αρχηγό) έως $N-1$. Στο Πρόγρ. 4.21 δείχνουμε ένα απλοϊκό παράδειγμα, όπου υπολογίζονται παράλληλα τα

```
#include <omp.h>

char *strings[10];        /* 10 pointers to strings (shared) */

int main() {
  int lengths[10];        /* also shared */

  read_strings(strings);   /* get the strings somehow */
  #pragma omp parallel num_threads(10)
  {                                          /* fork */
    int mine = omp_get_thread_num();         /* my id, private */
    lengths[mine] = strlen( strings[mine] ); /* shared */
  }                                          /* join */
  print_strings(lengths);  /* show results */
  return 0;
}
```

Πρόγραμμα 4.21 *Παράδειγμα παράλληλης περιοχής σε OpenMP.*

μήκη διαφορετικών συμβολοσειρών (strings). Προσέξτε τη γραμμή 1 η οποία πρέπει να βρίσκεται σε κάθε πρόγραμμα OpenMP. Η γραμμή 9 αποτελεί την οδηγία παραλληλοποίησης της περιοχής που ακολουθεί, η οποία περικλείεται μέσα σε άγκιστρα { ... }, δηλαδή οι γραμμές 11–12 του κώδικα. Η οδηγία parallel δημιουργεί μία ομάδα νημάτων, το πλήθος των οποίων καθορίζεται με τη φράση num_threads(). Αν δεν υπάρχει αυτή η φράση, το πόσα νήματα θα δημιουργηθούν εξαρτάται από το εκάστοτε σύστημα. Όλα τα νήματα εκτελούν τον κώδικα της παράλληλης περιοχής. Το καθένα μπορεί να βρει την ταυτότητά του με την κλήση omp_get_thread_num() (γραμμή 11). Η μεταβλητή mine είναι τοπική μέσα στην παράλληλη περιοχή και άρα *ιδιωτική* σε κάθε νήμα. Επομένως, το αποτέλεσμα της γραμμής 12 είναι ότι κάθε νήμα υπολογίζει το μήκος διαφορετικής συμβολοσειράς και αποθηκεύει το αποτέλεσμα σε διαφορετικό στοιχείο του πίνακα lengths. Το αρχικό νήμα είναι το μόνο που ζει μετά το πέρας της παράλληλης περιοχής, και εκτυπώνει τα αποτελέσματα.

Οι καθολικές μεταβλητές είναι κοινόχρηστες σε όλα τα νήματα που θα δημιουργηθούν. Στο OpenMP, όμως, υπάρχει η δυνατότητα να μπουν σε κοινή χρήση ακόμα και μεταβλητές που δεν είναι καθολικές, όπως ο πίνακας lengths στο Πρόγρ. 4.21. Γενικότερα, όποια μεταβλητή είναι δηλωμένη πριν την παράλληλη περιοχή, θεωρείται αυτόματα και κοινόχρηστη στα νήματα της παράλληλης περιοχής και στο νήμα-γονέα. Στην ορολογία του OpenMP αυτό ονομάζεται *έμμεσος καθορισμός* (implicitly determined) των *χαρακτηριστικών κοινοχρησίας* (sharing attributes) μίας μεταβλητής, δηλαδή αν πρέπει να είναι κοινόχρηστη ή όχι. Εκτός αυτού, δίνεται η δυνατότητα να καθοριστούν ρητά (explicitly) τα χαρακτηριστικά κοινοχρησίας με κατάλληλες φράσεις *δεδομένων*, και συγκεκριμένα:

- shared(x,y): οι μεταβλητές x και y θα είναι κοινόχρηστες ανάμεσα στα νήματα

της ομάδας που θα δημιουργηθεί (και του νήματος-γονέα).

- private(x,y): θα οριστούν νέες, ιδιωτικές μεταβλητές x και y στο κάθε νήμα της ομάδας που θα δημιουργηθεί.
- firstprivate(x,y): σαν την private(x,y), μόνο που επιπλέον θα αρχικοποιηθούν από την αντίστοιχη υπάρχουσα μεταβλητή του νήματος-γονέα.

Ως παράδειγμα, στον παρακάτω κώδικα:

```
1  a = b = c = 1;
2  #pragma omp parallel shared(a) private(b) firstprivate(c)
3  {
4    b++; c++;
5    if (omp_get_thread_num() == 1) {
6      a++;
7      printf("%d, %d, %d", a, b, c);
8    }
9  }
10 printf("%d, %d, %d", a, b, c);
```

η μεταβλητή b θα γίνει ιδιωτική στο κάθε νήμα, με απροσδιόριστη αρχική τιμή, ενώ η c θα γίνει ιδιωτική μεν, αλλά θα αρχικοποιηθεί στην τιμή της προϋπάρχουσας c του γονέα, δηλαδή στο 1. Επομένως, η εκτύπωση του νήματος 1 στη γραμμή 7 θα είναι 2, τυχαία και 2. Τελειώνοντας η παράλληλη περιοχή, το νήμα-αρχηγός θα επανέλθει στην πρότερή του κατάσταση (ως γονέας) και θα συνεχίσει με τον υπόλοιπο κώδικα. Προσέξτε, όμως, ότι οι ιδιωτικές μεταβλητές που είχε χειριστεί στην παράλληλη περιοχή έχουν πλέον χαθεί και στη γραμμή 11 χειρίζεται τις προϋπάρχουσες μεταβλητές. Επομένως, δεν θα δει αλλαγές στις b και c παρά μόνο στην a η οποία ήταν κοινόχρηστη. Έτσι, η εκτύπωση αυτή θα δώσει 2, 1, 1.

Εκτός από τις παραπάνω φράσεις δεδομένων, το OpenMP δίνει τη δυνατότητα να γίνουν αυτόματοι συνδυασμοί ή πράξεις μεταξύ των ιδιωτικών μεταβλητών των νημάτων κατά τον τερματισμό της ομάδας. Αυτές οι λειτουργίες ονομάζονται *υποβιβάσεις* (reductions). Ο λόγος είναι ότι από ένα σύνολο τιμών (δηλ. των μεταβλητών από τα νήματα) καταλήγουμε (υποβιβαζόμαστε) σε ένα απλό, βαθμωτό μέγεθος, όπως π.χ. το άθροισμά τους ή το γινόμενό τους. Το OpenMP παρέχει πληθώρα λειτουργιών υποβίβασης συμπεριλαμβανομένων του αθροίσματος, του γινομένου, της εύρεσης μεγίστης ή ελάχιστης τιμής κ.α. Η φράση συντάσσεται ως reduction(operation : variable), όπου operation είναι η πράξη που θέλουμε να γίνει και variable είναι η μεταβλητή του κάθε νήματος που θα συμμετέχει. Να ένας σύντομος (αν και όχι ιδιαίτερα αποδοτικός) τρόπος να υπολογίσουμε το άθροισμα των αριθμών 0 έως 20:

```
int sum;
#pragma omp parallel reduction(+: sum) num_threads(21)
{
```

```
  sum = omp_get_thread_num();
}
printf("sum = %d\n", sum);
```

Επειδή βρίσκεται σε φράση υποβίβασης, η μεταβλητή sum θα είναι *υποχρεωτικά* ιδιωτική σε κάθε νήμα. Όταν τελειώσει η παράλληλη περιοχή, οι ιδιωτικές μεταβλητές sum συνδυάζονται με την πράξη της υποβίβασης (πρόσθεση εδώ) και το τελικό αποτέλεσμα καταχωρείται στην προϋπάρχουσα sum του νήματος-γονέα. Στο συγκεκριμένο παράδειγμα, το νήμα i δίνει την τιμή i στη δική του sum και επομένως, στο τέλος της παράλληλης περιοχής θα αθροιστούν οι επιθυμητές τιμές.

4.9.2 Αμοιβαίος αποκλεισμός και συγχρονισμός

Το OpenMP παρέχει κλειδαριές (και μάλιστα δύο τύπων), μέσω παρόμοιων κλήσεων με τα Pthreads. Ο συνήθης τύπος κλειδαριών είναι ο omp_lock_t και η βασική χρήση τους γίνεται ως εξής:

```
omp_lock_t lck;
omp_init_lock(&lck);     /* initialize the lock */
omp_set_lock(&lck);      /* lock */
omp_unset_lock(&lck);    /* unlock */
```

Επιπλέον, υπάρχει και ο τύπος των *εμφωλευμένων κλειδαριών* (nested locks), όπου επιτρέπεται σε όποιον κατέχει την κλειδαριά, πριν βγει από την κρίσιμη περιοχή, να την ξανακλειδώσει όσες φορές επιθυμεί, αρκεί να την ξεκλειδώσει άλλες τόσες, ώστε να ελευθερωθεί πλήρως για τα υπόλοιπα νήματα.

Υπάρχει, όμως, πολύ πιο εύκολος τρόπος για την επίτευξη αμοιβαίου αποκλεισμού, μέσω της οδηγίας critical:

```
#pragma omp critical
{
  ...
}
```

Η οδηγία αυτή ορίζει την περιοχή του κώδικα που ακολουθεί ως κρίσιμη, και φροντίζει για τον απαραίτητο αμοιβαίο αποκλεισμό (συνήθως μέσω μιας κρυφής κλειδαριάς). Ο αμοιβαίος αποκλεισμός αυτός αφορά σε *όλες* τις κρίσιμες περιοχές του κώδικα και πιθανώς αυτό να δημιουργεί έντονο συναγωνισμό μεταξύ των νημάτων ακόμα και αυτών που ασχολούνται με άσχετα πράγματα. Για τον λόγο αυτό, η οδηγία critical μπορεί να πάρει προαιρετικά σε παρένθεση ένα όνομα. Μόνο όσα νήματα πέφτουν σε κρίσιμη περιοχή με το ίδιο όνομα θα συναγωνίζονται μεταξύ τους.

Όσον αφορά στον συγχρονισμό, το OpenMP παρέχει barriers με την ομώνυμη οδηγία:

```
#pragma omp barrier
```

Η οδηγία αυτή συγχρονίζει τα νήματα που ανήκουν στην ίδια παράλληλη ομάδα και είναι από τις λίγες οδηγίες του OpenMP που δεν έχουν περιοχή του κώδικα συνδεδεμένη μαζί τους. Μπορούν να μπουν σε όποιο σημείο επιθυμούμε συγχρονισμό, αλλά τοποθετούνται και αυτόματα από το OpenMP σε σημεία που κρίνεται σκόπιμο, όπως θα δούμε σε λίγο.

4.9.3 Διαμοιρασμός εργασίας

Αυτό που έχουμε δει μέχρι στιγμής είναι ότι το OpenMP αντικαθιστά κάποιες συνήθεις κλήσεις (π.χ. δημιουργίας νημάτων, προστασίας κρίσιμων περιοχών) με σύντομες οδηγίες `#pragma`. Αυτό από μόνο του είναι σημαντικό αλλά δεν είναι αρκετό για να ξεχάσει κανείς τα Pthreads. Η δύναμη του OpenMP γίνεται πιο φανερή, όταν τα νήματα μίας παράλληλης ομάδας μοιράζονται εργασία μεταξύ τους (*worksharing*).

Μέσα σε μία παράλληλη περιοχή, το OpenMP προσφέρει τρεις δομές διαμοιρασμού εργασίας. Η πρώτη επιτρέπει μόνο σε ένα νήμα της ομάδας να εκτελέσει την περιοχή που ακολουθεί. Η αντίστοιχη οδηγία συντάσσεται ως εξής:

```
#pragma omp single
{
  ...
}
```

Από όσα νήματα προσπαθήσουν να εκτελέσουν τη δεδομένη περιοχή, μόνο ένα (όποιο τυχαία φτάσει πρώτο) θα προλάβει να την εκτελέσει, ενώ τα υπόλοιπα θα την προσπεράσουν.

Η δεύτερη δομή, χωρίζει τον κώδικα της περιοχής σε τμήματα (sections), και αφήνει τα νήματα να τα εκτελέσουν, με όποια σειρά έρθουν:

```
#pragma omp sections
{
  #pragma omp section
  {
    ...
  }
  #pragma omp section
  {
    ...
  }
  ...
}
```

Κάθε υποπεριοχή `section` αποτελεί ανεξάρτητο κομμάτι της εργασίας και μπορεί να εκτελείται *ταυτόχρονα* με κάποιο άλλο. Ποιο νήμα θα εκτελέσει ποιο `section` είναι εντελώς τυχαίο. Το μόνο βέβαιο είναι ότι θα εκτελεστούν τελικά όλα.

Η τρίτη και σημαντικότερη δομή μοιράζει τις επαναλήψεις ενός βρόχου for:

```
#pragma omp for
```

```
for (var = init; var op bound; varincr) {
  ...
}
```

Εδώ γίνεται η υπόθεση ότι οι επαναλήψεις του βρόχου είναι *ανεξάρτητες μεταξύ τους* και δεν επηρεάζει η μία τους υπολογισμούς που κάνει η άλλη· το OpenMP δεν το ελέγχει αυτό, είναι ευθύνη του προγραμματιστή. Ο βρόχος υποχρεωτικά πρέπει να συντάσσεται με συγκεκριμένη μορφή. Το πρώτο του τμήμα πρέπει να έχει μία καταχώρηση στη μεταβλητή-μετρητή του βρόχου (`var`), η οποία της δίνει την αρχική της τιμή. Το δεύτερο τμήμα του πρέπει να έχει μία μόνο συνθήκη από την οποία προκύπτει η τελική τιμή του μετρητή (`bound`). Ο τελεστής `op` είναι ένας από τους τελεστές σύγκρισης `<`, `<=`, `>` ή `>=`. Τέλος, το τρίτο μέρος του (`varincr`) θα πρέπει να είναι μία καταχώρηση προς τον μετρητή, ώστε να προκύπτει καθαρά το βήμα κατά το οποίο αυξάνεται ή μειώνεται σε κάθε επανάληψη. Επιτρέπονται, επομένως, εκφράσεις όπως `var++`, `var = var - 2*n` κλπ.

Πριν αναλύσουμε με λεπτομέρεια την ακριβή λειτουργία της οδηγίας `for`, θα πρέπει να σημειώσουμε κάποια χαρακτηριστικά που είναι κοινά και στις τρεις δομές διαμοιρασμού εργασίας:

- Όλα τα νήματα της ομάδας πρέπει υποχρεωτικά να συναντήσουν την περιοχή διαμοιρασμού εργασίας, ακόμα κι ας μην τύχει να εκτελέσουν κάτι σε αυτήν.

- Στο τέλος της περιοχής υπονοείται συγχρονισμός των νημάτων. Είναι σαν να υπάρχει μία οδηγία `barrier`, αμέσως μετά την οδηγία διαμοιρασμού εργασίας. Στην περίπτωση που ο προγραμματιστής κρίνει ότι δεν είναι απαραίτητος ο συγχρονισμός, τού δίνεται η δυνατότητα να τον αποφύγει (π.χ. για λόγους ταχύτητας). Αυτό γίνεται με το να συμπεριλάβει τη φράση `nowait` στην οδηγία διαμοιρασμού εργασίας.

- Παρέχονται οι φράσεις δεδομένων `private()` και `firstprivate()`, που είδαμε και στην οδηγία `parallel`, για δημιουργία ιδιωτικών αντιγράφων των σημειωμένων μεταβλητών. Ανάλογα με τη δομή, παρέχονται και κάποιες επιπλέον φράσεις δεδομένων. Στην οδηγία `for`, η μεταβλητή-μετρητής θεωρείται αυτόματα `private()`.

4.9.4 Υπολογισμός του π με OpenMP

Με βάση όσα είδαμε μέχρι στιγμής, μπορούμε να υλοποιήσουμε την πρώτη μας αξιόλογη εφαρμογή. Ας δούμε, λοιπόν, πώς μπορούμε να υπολογίσουμε το π με OpenMP. Ανατρέξτε στην Ενότητα 4.6.1 για να θυμηθείτε τόσο τον αρχικό σειριακό κώδικα όσο και τον παραλληλοποιημένο κώδικα με νήματα POSIX (Πρόγρ. 4.14). Η πρώτη μας προσπάθεια για παραλληλοποίηση του προγράμματος με OpenMP δίνεται στο Προγρ. 4.22. Είναι σωστή;

```
1  #include <omp.h>
2  #define N 10000000
3
4  int main() {
5    int    i;
6    double pi = 0.0, mysum = 0.0, W = 1.0/N;
7
8    #pragma omp parallel
9    {
10     #pragma omp for
11       for (i=0; i < N; i++) {
12         mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
13       }
14     #pragma omp critical
15     {
16       pi += mysum;
17     }
18   }
19   printf("pi = %.10lf\n", pi);
20 }
```

Πρόγραμμα 4.22 *Πρώτη προσπάθεια για παραλληλοποίηση του υπολογισμού του π με χρήση OpenMP. (sas-omp-pi1.c)*

Αρχικά δημιουργούμε τα απαραίτητα νήματα με την οδηγία parallel στη γραμμή 8. Κάθε νήμα θα συναντήσει τη δομή διαμοίρασης εργασίας (οδηγία for στη γραμμή 10). Επομένως, τα νήματα θα μοιραστούν τις επαναλήψεις του βρόχου for που ακολουθεί. Αμέσως μετά, κάθε νήμα θα προσθέσει, μέσω αμοιβαίου αποκλεισμού (οδηγία critical στη γραμμή 14), τη συνεισφορά του στο pi. Με το πέρας της παράλληλης περιοχής, το αρχικό νήμα είναι το μόνο που θα συνεχίσει την εκτέλεσή του και θα τυπώσει το αποτέλεσμα. Η λογική και η δομή του προγράμματος είναι πολύ σωστή—εκτός από μία λεπτομέρεια, που αποτελεί και το πιο συνηθισμένο λάθος όταν ξεκινά κανείς την προγραμματιστική του διαδρομή με το OpenMP: τα χαρακτηριστικά κοινοχρησίας των μεταβλητών. Όπως είπαμε, όποιες μεταβλητές έχουν οριστεί πριν από μία παράλληλη περιοχή, είναι εξ ορισμού κοινόχρηστες μεταξύ των νημάτων, εκτός και αν έχει ρητά ζητηθεί κάτι άλλο μέσω κατάλληλων φράσεων δεδομένων. Επομένως, τόσο η μεταβλητή pi, όσο και οι mysum, i, W, θα είναι κοινόχρηστες. Η W δεν τροποποιείται και άρα δεν δημιουργεί κάποιο θέμα. Η i, όπως γνωρίζουμε, θα γίνει υποχρεωτικά ιδιωτική μέσα στην περιοχή του βρόχου for, ενώ εξω από αυτή δεν τροποποιείται, και επομένως και με αυτή δεν υπάρχει πρόβλημα. Η mysum, όμως, τροποποιείται από όλα τα νήματα στη γραμμή 12, και αυτό δεν ήταν στον σχεδιασμό μας. θα έπρεπε να είναι ιδιωτική μεταβλητή στο κάθε νήμα, και μάλιστα αρχικοποιημένη στο 0, όπως η προϋπάρχουσα mysum του αρχικού νήματας (γραμμή 6). Επομένως, θα έπρεπε να είχε δηλωθεί

```
1  #include <omp.h>
2  #define N 10000000
3
4  int main() {
5    int    i;
6    double pi = 0.0, mysum = 0.0, W = 1.0/N;
7
8    #pragma omp parallel firstprivate(mysum, W)
9    {
10     #pragma omp for nowait
11       for (i=0; i < N; i++)
12         mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
13     #pragma omp critical
14       pi += mysum;
15   }
16   printf("pi = %.10lf\n", pi);
17 }
```

Πρόγραμμα 4.23 *Δεύτερη εκδοχή παραλληλοποίησης του υπολογισμού του π με χρήση OpenMP. (sas-omp-pi2.c)*

ως firstprivate στην παράλληλη περιοχή.

Η ορθή εκδοχή δίνεται στο Προγρ. 4.23, με το σωστό χαρακτηριστικό κοινοχρησίας για μεταβλητές mysum και W, αν και η W δεν είχε πρόβλημα και ως κοινόχρηστη. Στο πρόγραμμα αυτό, επίσης, φαίνεται ότι οι περιοχές που ακολουθούν μία οδηγία, δεν χρειάζεται να περικλείονται σε άγκιστρα { ... }, εφόσον περιέχουν μόνο μία εντολή, όπως συμβαίνει και με όλα τα μπλοκ στη γλώσσα C. Έτσι, από την περιοχή της οδηγίας critical αφαιρέθηκαν τα άγκιστρα, μιας και αποτελείται μόνο από την εντολή στη γραμμή 15. Τέλος, παρατηρήστε ότι μετά το for υπονοείται barrier. Εδώ όμως, δεν υπάρχει κανένας λόγος να συγχρονιστούν τα νήματα και άρα να υποστούμε το χρονικό αυτό κόστος. Επομένως, για λόγους βελτιστοποίησης, προστέθηκε η φράση nowait στην οδηγία for.

Στο Προγρ. 4.23 παρατηρήστε ότι κάθε νήμα υπολογίζει μία ιδιωτική mysum, και όλες οι mysum προστίθενται μεταξύ τους για να υπολογιστεί η τιμή του pi. Αυτή είναι κλασική περίπτωση *υποβίβασης* αθροίσματος. Ο κώδικας, επομένως, μπορεί να απλοποιηθεί κι άλλο, όπως φαίνεται στο Πρόγρ. 4.24. Πια, δεν υπάρχει mysum, και το pi αποτελεί μεταβλητή υποβίβασης (άρα ιδιωτική). Κάθε νήμα υπολογίζει το δικό του pi με βάση τις επαναλήψεις που του αντιστοιχούν. Στο τέλος της παράλληλης περιοχής προστίθενται όλα τα ιδιωτικά pi, και το αποτέλεσμα καταχωρείται στην προϋπάρχουσα μεταβλητή pi του αρχικού νήματος. Όλο αυτό το αναλαμβάνει να το φέρει εις πέρας το OpenMP, οπότε δεν χρειάζεται να ασχοληθούμε εμείς με τις λεπτομέρειες των πράξεων και του αμοιβαίου αποκλεισμού.

Στο Πρόγρ. 4.24 η παράλληλη περιοχή περιέχει, πλέον, μόνο ένα βρόχο for. Είναι τόσο συνηθισμένη αυτή η περίπτωση στην πράξη, ώστε το OpenMP επιτρέπει το συνδυασμό

```
1  #include <omp.h>
2  #define N 10000000
3
4  int main() {
5    int    i;
6    double pi, W = 1.0/N;
7
8    #pragma omp parallel firstprivate(W) reduction(+:pi)
9      #pragma omp for nowait
10       for (i=0; i < N; i++)
11         pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
12   printf("pi = %.10lf\n", pi);
13 }
```

Πρόγραμμα 4.24 *Τρίτη εκδοχή παραλληλοποίησης του υπολογισμού του π με χρήση OpenMP. (sas-omp-pi3.c)*

```
1  #define N 10000000
2
3  int main() {
4    int    i;
5    double pi, W = 1.0/N;
6
7    #pragma omp parallel for firstprivate(W) reduction(+:pi)
8      for (i=0; i < N; i++)
9        pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
10   printf("pi = %.10lf\n", pi);
11 }
```

Πρόγραμμα 4.25 *Τελική εκδοχή παραλληλοποίησης του υπολογισμού του π με χρήση OpenMP. (sas-omp-pi4.c)*

των δύο οδηγιών `parallel` και `for` σε μια γραμμή[8]. Αυτό μας οδηγεί στην τελική έκδοση του κώδικα, στο Πρόγρ. 4.25. Τα νήματα δημιουργούνται μόνο για να μοιραστούν τις επαναλήψεις του βρόχου `for`, και καταστρέφονται αμέσως μετά. Η φράση `nowait` είναι περιττή, καθώς στην περίπτωση αυτή το σύστημα δεν εισάγει barrier μιας και τα νήματα καταστρέφονται αμέσως μετά το βρόχο.

Θα κλείσουμε την ενότητα αυτή με μια σημαντική παρατήρηση. Ο τελικός κώδικας στο Πρόγρ. 4.25 είναι *ίδιος* με τον σειριακό, πλην της γραμμής 8. Αγνοώντας τη γραμμή αυτή, αυτό που μένει είναι ένας ορθός σειριακός υπολογισμός του π. Με την εισαγωγή μίας μόλις οδηγίας του OpenMP παραλληλοποιήσαμε ένα πρόγραμμα, το οποίο όταν το παραλληλοποιούσαμε με νήματα (Πρόγρ. 4.14) θέλαμε περίπου τριπλάσιες γραμμές κώ-

[8] Η μόνη άλλη συνδυασμένη (combined) οδηγία είναι η `#pragma omp parallel sections`.

δικα, και μάλιστα κώδικα εντελώς διαφορετικού από τον σειριακό. Εδώ ακριβώς βρίσκεται η δύναμη του OpenMP, και ο λόγος για τον οποίον έχει γίνει ο *de facto* τρόπος παράλληλου προγραμματισμού στο μοντέλο κοινόχρηστου χώρου διευθύνσεων: δεν αλλάζει το σειριακό πρόγραμμα, παρά μόνο απαιτεί την προσθήκη απλών οδηγιών σε μερικά σημεία, προκειμένου να αναλάβει τις λεπτομέρειες της παραλληλοποίησής του.

4.9.5 Βρόχοι με την οδηγία for

Η οδηγία for, όπως είδαμε, μοιράζει τις επαναλήψεις του αντίστοιχου βρόχου μεταξύ των νημάτων μίας παράλληλης ομάδας. Ο τρόπος που τις μοιράζει, όμως, εξαρτάται από τον εκάστοτε μεταφραστή του OpenMP. Ενώ αυτό σε πολλές περιπτώσεις είναι αρκετό, σε άλλες δεν δίνει τα ιδεώδη αποτελέσματα. Για αυτόν το λόγο, δίνεται η δυνατότητα στους προγραμματιστές να επιλέξουν τον τρόπο με τον οποίο θα γίνει η διαμοίραση. Η επιλογή γίνεται με την εξής φράση στην οδηγία for:

```
schedule(schedtype, chunksize)
```

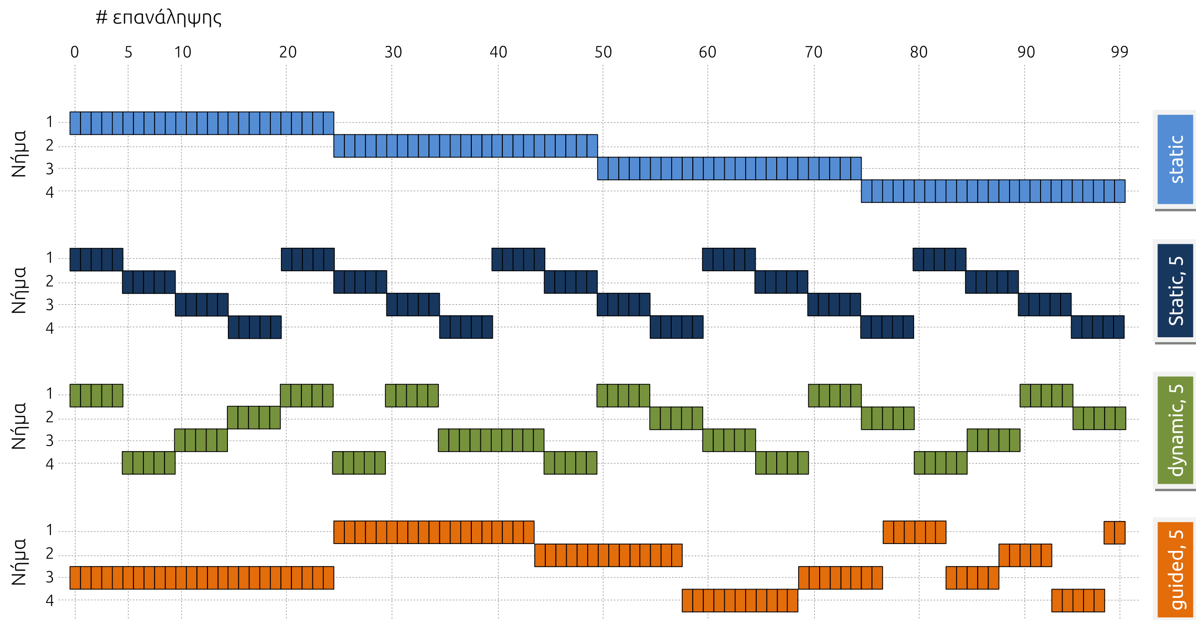
Όπου schedtype είναι ο τύπος της διαμοίρασης, ενώ το προαιρετικό chunksize καθορίζει το μέγεθος των τμημάτων (chunks), δηλαδή το πλήθος των επαναλήψεων ανά τμήμα, που θα δίνονται κάθε φορά. Υπάρχουν, βασικά, τρεις διαφορετικοί τύποι διαμοίρασης: static, dynamic και guided.

- Ο τύπος static προκαθορίζει τις επαναλήψεις που θα πάρει το κάθε νήμα. Εφόσον δεν δοθεί chunksize, μοιράζει τις επαναλήψεις όσο πιο ισόποσα γίνεται, κάνοντας τμηματική δρομολόγηση. Αν δοθεί chunksize, κάνει δρομολόγηση με άλματα, όπου κάθε φορά, αντί να δίνει 1 επανάληψη στο κάθε νήμα, δίνει chunksize συνεχόμενες επαναλήψεις.

- Ο τύπος dynamic μοιράζει τις επαναλήψεις ουσιαστικά κάνοντας χρήση της τεχνικής της αυτοδρομολόγησης. Αφήνει τα νήματα να συναγωνίζονται στο ποιο θα πάρει το επόμενο τμήμα να εκτελέσει, όπου το κάθε τμήμα αποτελείται από chunksize συνεχόμενες επαναλήψεις. Αν δεν δοθεί chunksize, το κάθε τμήμα θα έχει ακριβώς 1 επανάληψη.

- Ο τύπος guided κάνει χρήση της τεχνικής της αυτοδρομολόγησης, επίσης. Η διαφορά με το dynamic είναι ότι το επόμενο τμήμα που θα δοθεί προς εκτέλεση είναι μικρότερο από το προηγούμενο που δόθηκε. Τα αρχικά τμήματα είναι σχετικά μεγάλα, και όσο περνά η ώρα μειώνονται σε μέγεθος, αλλά ποτέ δεν δίνονται μικρότερα από chunksize (εκτός ίσως από το τελευταίο τμήμα).

Σε συνθήκες όπου οι επεξεργαστές έχουν ίδιες ταχύτητες, περίπου τον ίδιο φόρτο, και οι επαναλήψεις μεταξύ τους έχουν παρόμοια εργασία να κάνουν, η δρομολόγηση static



Σχήμα 4.26 *Διαμοίραση επαναλήψεων στο OpenMP με διαφορετικούς τύπους δρομολόγησης*

είναι συνήθως η προτιμότερη. Σε αντίθετη περίπτωση, θα αντληθούν καλύτερες επιδόσεις με κάποια από τις άλλες δύο στρατηγικές. Η δρομολόγηση dynamic είναι περισσότερο για περιπτώσεις με μεγάλες διακυμάνσεις είτε στις ταχύτητες των επεξεργαστών είτε στο φόρτο εργασίας ανά επανάληψη. Όσο μεγαλύτερο είναι το chunksize τόσο λιγότερη είναι και η αποτελεσματικότητα της αυτοδρομολόγησης, ενώ όσο μικρότερο είναι τόσο πιο πολύ αυξάνεται ο συναγωνισμός ανάμεσα στα νήματα. Η δρομολόγηση guided είναι αρκετές φορές ένας καλός συμβιβασμός, προκειμένου να μην υπάρξει μεγάλος συναγωνισμός καθ' όλη τη διάρκεια της εκτέλεσης.

Ο τρόπος με τον οποίο μοιράζονται οι επαναλήψεις φαίνεται και γραφικά στο Σχ. 4.26. Στο συγκεκριμένο παράδειγμα, έχουμε υποθέσει ότι υπάρχουν 4 νήματα, τα οποία θα μοιραστούν συνολικά 100 επαναλήψεις. Το chunksize, όπου χρειάζεται, έχει οριστεί σε 5 επαναλήψεις.

Αξίζει να σημειώσουμε ότι το OpenMP προσφέρει και έναν ακόμα τύπο δρομολόγησης, το schedule(runtime). Δεν πρόκειται για κάποια νέα πολιτική διαμοίρασης επαναλήψεων, απλά για απόφαση να επιλεχτεί ο τελικός τρόπος διαμοίρασης την ώρα της εκτέλεσης του προγράμματος. Συγκεκριμένα, δίνοντας την κατάλληλη τιμή στη μεταβλητή περιβάλλοντος OMP_SCHEDULE, ο χρήστης μπορεί να καθορίσει οποιαδήποτε από τις τρεις πολιτικές δρομολόγησης είδαμε παραπάνω, χωρίς να χρειαστεί να τροποποιήσει το πρόγραμμά του και να το επαναμεταφράσει. Στην πράξη, η χρήση της φράσης schedule(runtime) γίνεται συνήθως για λόγους πειραματισμού, προκειμένου να διαπιστωθεί ποιος τύπος schedule αποδίδει καλύτερα.

4.9.6 Άλλα χαρακτηριστικά του OpenMP

Το OpenMP διαθέτει κάποιες επιπλέον οδηγίες αλλά και πληθώρα άλλων ευκολιών που μπορεί να φανούν χρήσιμες σε κάποιες περιπτώσεις. Μερικές θα τις δούμε συνοπτικά εδώ, αλλά ο αναγνώστης παροτρύνεται να ανατρέξει στις βιβλιογραφικές πηγές στο τέλος του κεφαλαίου για μια πληρέστερη μελέτη.

Το OpenMP επιτρέπει στα νήματα να έχουν ιδιωτικές καθολικές μεταβλητές με την οδηγία:

```
#pragma omp threadprivate(var1, var2, ...)
```

όπου οι μεταβλητές var1, var2, κλπ. είναι ήδη δηλωμένες, πριν την οδηγία. Αξίζει να σημειώσουμε ότι κάτι τέτοιο είναι δυνατόν να γίνει και με τα Pthreads, αλλά μέσω ενός πιο πολύπλοκου μηχανισμού που ονομάζεται thread specifics.

Το OpenMP επιτρέπει, επίσης, ένθετες παράλληλες περιοχές. Με άλλα λόγια, ένα νήμα που ανήκει σε μία ομάδα, μπορεί να δημιουργήσει μία δική του ομάδα νημάτων μέσω μίας οδηγίας parallel. Αυτό ονομάζεται *εμφωλευμένος παραλληλισμός* (nested parallelism) και είναι μερικές φορές πολύ χρήσιμος, αλλά γενικά μπορεί εύκολα να οδηγήσει σε υπερεγγραφή (oversubscription) των επεξεργαστών και συνεπακόλουθα, κατακόρυφη μείωση των επιδόσεων. Για αυτόν τον λόγο θεωρείται δύσκολος στη χρήση του και δεν υποστηρίζεται πλήρως από πολλούς μεταφραστές. Το πρότυπο προβλέπει ότι όταν συναντηθεί μία εμφωλευμένη παράλληλη περιοχή, ο μεταφραστής επιτρέπεται να δημιουργήσει ομάδα του ενός μόλις νήματος (και άρα να εκτελεστεί σειριακά), ώστε να αποφευχθεί η ανεξέλεγκτη δημιουργία νημάτων. Ο προγραμματιστής μπορεί να ενεργοποιήσει ή όχι τον εμφωλευμένο παραλληλισμό μέσω συγκεκριμένης μεταβλητής περιβάλλοντος (OMP_NESTED) ή προγραμματιστικών κλήσεων (omp_set_nested()).

Γενικότερα, το πλήθος των νημάτων που θα έχει μία οποιαδήποτε ομάδα αποφασίζεται μέσα από μία σειρά από ελέγχους. Εφόσον έχει δοθεί η φράση num_threads(), αυτή θα καθορίσει το πλήθος των νημάτων. Εάν δεν έχει δοθεί, το πλήθος καθορίζεται από την κλήση omp_set_num_threads(). Εάν δεν έχει γίνει η κλήση αυτή, εξετάζεται η μεταβλητή περιβάλλοντος OMP_NUM_THREADS. Εφόσον και αυτή δεν έχει οριστεί, ο μεταφραστής χρησιμοποιεί ένα προκαθορισμένο πλήθος. Ο προγραμματιστής μπορεί, τέλος, να δώσει την ελευθερία στον μεταφραστή να ρυθμίζει αυτός το τελικό πλήθος των νημάτων, μέσω της μεταβλητής περιβάλλοντος OMP_DYNAMIC ή της κλήσης omp_set_dynamic().

Τέλος, οι κλήσεις βιβλιοθήκης που παρέχει το OpenMP είναι πολύ χρήσιμες για λήψη πληροφοριών ή ενεργοποίηση επιλογών κατά τη διάρκεια εκτέλεσης. Υπάρχουν ρουτίνες που επιστρέφουν την ταυτότητα του νήματος, το πλήθος νημάτων της ομάδας, την ταυτότητα του γονέα ή κάποιου προγόνου, κλπ. Αντίστοιχα, υπάρχει πληθώρα άλλων μεταβλητών περιβάλλοντος που καθορίζουν πολλά χαρακτηριστικά της εκτέλεσης, όπως π.χ. πώς θα ανατεθούν τα νήματα στους πυρήνες, ή ποιο είναι το επιθυμητό μέγεθος στοίβας για κάθε νήμα.

4.10 Παραλληλοποίηση «ακανόνιστων» εφαρμογών

Μέχρι στιγμής έχουμε δει παραδείγματα εφαρμογών των οποίων η εκτέλεση κυριαρχείται από επαναληπτικούς βρόχους. Πολλές επιστημονικές εφαρμογές, με ιδιαίτερα μεγάλες υπολογιστικές απαιτήσεις, εμπίπτουν στην κατηγορία αυτή, και το μεγαλύτερο τμήμα του χρόνου εκτέλεσης καταναλώνεται σε τέτοιους βρόχους. Αυτός είναι και ο λόγος που η παραλληλοποίηση ξεκινά σχεδόν πάντα από τους βρόχους for. Το OpenMP είναι ιδιαίτερα επιτυχημένο σε αυτές ακριβώς τις περιπτώσεις.

Υπάρχει όμως και πληθώρα άλλων εφαρμογών που είτε δεν καταναλώνουν τον χρόνο τους σε βρόχους for είτε στηρίζονται σε αναδρομικές κλήσεις, είτε γενικά έχουν ακατάστατους υπολογισμούς οι οποίοι δεν είναι εκ των προτέρων γνωστοί, και προκύπτουν κατά τη διάρκεια της εκτέλεσης. Όλες αυτές οι εφαρμογές είναι γνωστές και ως *ακανόνιστες* (irregular), και η αποδοτική παραλληλοποίησή τους δεν είναι πολύ εύκολη.

Από την έκδοση 3.0 και μετά, το OpenMP διαθέτει μία νέα οδηγία, η οποία δίνει τη δυνατότητα παραλληλοποίησης μεγάλης κατηγορίας εφαρμογών που δεν στηρίζονται σε βρόχους for. Ένα χαρακτηριστικό παράδειγμα είναι οι λεγόμενες εφαρμογές που στηρίζονται στο κυνήγι δεικτών (pointer chasing), όπως π.χ. όταν η βασική δομή δεδομένων είναι μία συνδεδεμένη λίστα. Φανταστείτε ότι θέλουμε να κάνουμε μία παρόμοια εργασία σε κάθε κόμβο αυτής της λίστας, και επειδή ο κάθε κόμβος είναι ανεξάρτητος από τον άλλο, θέλουμε να κάνουμε όλα παράλληλα. Επειδή τα δεδομένα δεν βρίσκονται σε συνεχόμενες θέσεις μνήμης, και για να βρούμε πού είναι ο επόμενος κόμβος πρέπει πρώτα να επισκεφτούμε τον προηγούμενό του, η διάσχιση της λίστας δεν μπορεί να εκφραστεί ως ένας βρόχος for με προκαθορισμένα όρια, και, επομένως, δεν μπορούμε να μοιράσουμε τους κόμβους στα νήματα.

Μιά εργασία (task) στο OpenMP είναι ένα κομμάτι κώδικα που πρέπει να εκτελεστεί κάποτε, από κάποιο νήμα, και όχι απαραίτητα από αυτό που το συνάντησε. Δεν υπάρχει, δηλαδή, προκαθορισμένο νήμα που θα το εκτελέσει, ούτε και προκαθορισμένος χρόνος που πρέπει να γίνει η εκτέλεση. Το νήμα που συναντά αυτό το τμήμα του κώδικα το «ετοιμάζει» για εκτέλεση και το αφήνει στην άκρη μέχρι να βρεθεί κάποιος να το εκτελέσει. Η οδηγία συντάσσεται ως εξής:

```
#pragma omp task
{
  ...
}
```

Η περιοχή που ακολουθεί την οδηγία είναι ο κώδικας της εργασίας που πρέπει να εκτελεστεί. Μόλις ένα νήμα συναντήσει την οδηγία, επιτελείται η *δημιουργία* της εργασίας. Το κρίσιμο στοιχείο είναι ότι, όταν έρθει η ώρα να εκτελεστεί, οι μεταβλητές που χρησιμοποιούνται μέσα στην περιοχή πρέπει να έχουν τις τιμές που είχαν κατά τη δημιουργία της εργασίας. Επομένως, τη στιγμή που δημιουργείται μία εργασία, το σύστημα παίρνει

και ένα στιγμιότυπο του περιβάλλοντος δεδομένων (μεταβλητών). Ο κώδικας μαζί με το στιγμιότυπο αυτό, αποτελούν την ολοκληρωμένη εργασία (task) η οποία φυλάσσεται από το σύστημα μέχρι κάποιο νήμα να την παραλάβει και να την εκτελέσει.

Δεν υπάρχει κάποια πρόβλεψη για το πότε και πώς θα εκτελεστούν οι εργασίες. Είναι όμως σίγουρο ότι θα έχει ολοκληρωθεί η εκτέλεσή τους σε συγκεκριμένα σημεία του κώδικα:

- Στο τέλος μίας παράλληλης περιοχής, είναι εγγυημένο ότι θα έχουν εκτελεστεί όλες οι εργασίες που έχουν δημιουργήσει τα νήματα της ομάδας.
- Μετά από έναν barrier, είναι εγγυημένο ότι θα έχουν εκτελεστεί όλες οι εργασίες που έχουν δημιουργήσει τα νήματα μίας παράλληλης ομάδας μέχρι εκείνη τη στιγμή.
- Μετά από μία οδηγία `#pragma omp taskwait` είναι εγγυημένο ότι θα έχουν εκτελεστεί όλες οι εργασίες που έχει δημιουργήσει το νήμα που συνάντησε την οδηγία αυτή.

Στη δεύτερη περίπτωση τα νήματα συγχρονίζονται και επιπλέον εκτελούν και όποια εργασία έχει δημιουργηθεί από την ομάδα και δεν έχει εκτελεστεί ακόμα. Στην τρίτη περίπτωση το νήμα δεν συνεχίζει, αν δεν ολοκληρωθεί η εκτέλεση των εργασιών που δημιούργησε το ίδιο.

Όπως και στις παράλληλες περιοχές, έτσι και στις εργασίες, τα χαρακτηριστικά κοινοχρησίας των μεταβλητών που χρησιμοποιούνται καθορίζονται είτε έμμεσα από τον μεταφραστή είτε άμεσα από τον προγραμματιστή. Για μεταβλητές με έμμεσο καθορισμό (όταν δηλαδή, ο προγραμματιστής δεν έχει κάνει ρητό προσδιορισμό), ο μεταφραστής χρησιμοποιεί τον εξής κανόνα:

- Αν μία μεταβλητή της εργασίας είναι κοινόχρηστη μεταξύ των νημάτων στην περιοχή μέσα στην οποία περικλείεται η εργασία, τότε κατηγοριοποιείται ως κοινόχρηστη.
- Σε οποιαδήποτε άλλη περίπτωση η μεταβλητή θεωρείται ότι είναι `firstprivate`.

Επειδή όμως, ο κανόνας αυτός αφήνει πολλές φορές περιθώρια για λανθασμένες εκτιμήσεις, συνίσταται να καθορίζονται ρητά όλες οι μεταβλητές που συμμετέχουν στην εργασία ως κοινόχρηστες (φράση `shared()`) ή μη (φράσεις `private()` και `firstprivate()`). Το σημαντικό σημείο είναι ότι στο στιγμιότυπο δεδομένων που αποθηκεύεται μαζί με τον κώδικα της εργασίας *συμμετέχουν μόνο οι μεταβλητές `firstprivate()`*. Οι κοινόχρηστες θα χρησιμοποιηθούν με την τιμή που θα έχουν τη στιγμή της εκτέλεσης της εργασίας.

4.10.1 Διάσχιση λίστας

Με βάση τα παραπάνω, ας δούμε πώς θα παραλληλοποιούσαμε ένα πρόβλημα με κυνήγι δεικτών μέσω εργασιών του OpenMP. Ο σειριακός κώδικας είναι ο εξής:

```
for (elem = list->start; elem != NULL; elem = elem->next)
  workon(elem);
```

Η ιδέα είναι να δημιουργηθούν τόσες εργασίες όσα είναι και τα στοιχεία της λίστας και να εκτελεστούν από τα νήματα μιας παράλληλης ομάδας. Για να γίνει αυτό, αρκεί να «ντύσουμε» την workon() με μία οδηγία task:

```
  #pragma omp task
    workon(elem);
```

Επειδή δεν μπορούν να μοιραστούν οι επαναλήψεις του βρόχου for, μόνο ένα νήμα θα πρέπει να τις αναλάβει και να δημιουργήσει όλες τις εργασίες. Αυτό μπορεί να γίνει εύκολα μέσα από μία περιοχή single. Επομένως, δεν θα έχουμε παραλληλισμό κατά τη δημιουργία των εργασιών, παρά μόνο κατά την εκτέλεσή τους που είναι πολύ πιο χρονοβόρα (θεωρώντας ότι η εργασία που επιτελεί η workon() δεν είναι αμελητέα).

Μετά από την παραπάνω ανάλυση, ο κώδικας που προκύπτει είναι ο εξής:

```
#pragma omp parallel
{
  #pragma omp single
  {
    for (elem = list->start; elem != NULL; elem = elem->next)
      #pragma omp task
        workon(elem);
  }
}
```

Δημιουργείται μία παράλληλη ομάδα, και ένα από όλα τα νήματα θα εκτελέσει το βρόχο for, δημιουργώντας μία εργασία ανά κόμβο. Στο τέλος του single υπονοείται barrier και επομένως, εκεί θα έχουν ολοκληρώσει την εκτέλεσή τους όλες οι εργασίες που θα δημιουργηθούν. Ακόμα και τη φράση nowait να βάζαμε στην οδηγία single, το τέλος της περιοχής parallel είναι σημείο που επίσης εξασφαλίζει την εκτέλεση των εργασιών. Το πρόγραμμα έτσι όπως το έχουμε, όμως, έχει ένα πρόβλημα. Μία ακόμα φορά, πρέπει να είμαστε προσεκτικοί με τα χαρακτηριστικά κοινοχρησίας των μεταβλητών. Στον παραπάνω κώδικα, η μεταβλητή elem καθορίζεται έμμεσα από τον μεταφραστή ως κοινόχρηστη. Ο λόγος είναι ότι έχει οριστεί έξω από την παράλληλη περιοχή και άρα, είναι κοινόχρηστη ανάμεσα στα νήματα. Η δομή της εργασίας είναι λεκτικά εμφωλευμένη μέσα στη δομή single, η οποία είναι μέσα στη δομή parallel, και έτσι ο μεταφραστής διαπιστώνει την κοινοχρησία της μεταβλητής. Επομένως, είναι απαραίτητος ο ρητός καθορισμός της ως firstprivate(), ώστε η κάθε εργασία να «θυμάται» τον κόμβο για τον οποίο θα εργαστεί όταν έρθει η ώρα της εκτέλεσης. Ο τελικός κώδικας δίνεται στο Πρόγρ. 4.27.

```
#pragma omp parallel
{
  #pragma omp single
  {
    for (elem = list->start; elem != NULL; elem = elem->next)
      #pragma omp task firstprivate(elem)
        workon(elem);
  }
}
```

Πρόγραμμα 4.27 *Διάσχιση λίστας με εργασίες του OpenMP.*

4.10.2 Αναδρομικοί υπολογισμοί

Μία δεύτερη κατηγορία εφαρμογών που μπορεί να παραλληλοποιηθεί με εργασίες είναι αυτές που στηρίζονται στην αναδρομή. Ας πάρουμε ως παράδειγμα, τον αναδρομικό υπολογισμό των γνωστών αριθμών Fibonacci, οι οποίοι ορίζονται ως εξής: ο n-οστός αριθμός, $F(n)$, δίνεται από τη σχέση:

$$F(n) = F(n-1) + F(n-2), n \geq 2,$$

με $F(0) = 0$ και $F(1) = 1$. Για το συγκεκριμένο πρόβλημα υπάρχουν και ταχύτεροι αλγόριθμοι, αλλά ας υποθέσουμε ότι θέλουμε να χρησιμοποιήσουμε αναδρομική λύση. Ο σειριακός κώδικας είναι κλασικός:

```
int fib(int n) {
  int x, y;

  if (n < 2) return n;
  x = fib(n-1);
  y = fib(n-2);
  return (x+y);
}
```

Το ζήτημα είναι να παραλληλοποιηθεί ο παραπάνω κώδικας, και θα το κάνουμε με χρήση των εργασιών του OpenMP.

Χωρίς να μπούμε σε όλες τις λεπτομέρειες, υποτίθεται ότι κάπου αλλού στο πρόγραμμά μας έχει δημιουργηθεί μία ομάδα νημάτων και κάποιο από όλα έχει κάνει κλήση στην `fib()`, προκειμένου να υπολογιστεί παράλληλα ο n-οστός αριθμός Fibonacci. Εδώ δεν υπάρχει βρόχος for, και οι εργασίες είναι η μόνη εναλλακτική προσέγγιση. Η σκέψη είναι σχετικά απλή: θα δημιουργήσουμε μία διαφορετική εργασία για κάθε κλήση της `fib()`, στηριζόμενοι στην ύπαρξη πολλαπλών νημάτων για την εκτέλεσή τους:

```
int fib(int n) {
  int x, y;
```

```
1  int fib(int n) {
2    int x, y;
3
4    if (n < 2) return n;
5    #pragma omp task shared(x) firstprivate(n)
6      x = fib(n-1);
7    #pragma omp task shared(y) firstprivate(n)
8      y = fib(n-2);
9    #pragma omp taskwait
10   return (x+y);
11 }
```

Πρόγραμμα 4.28 *Αναδρομικός υπολογισμός αριθμών Fibonacci με εργασίες του OpenMP, πρώτη έκδοση. (sas-omp-fib1.c)*

```
3
4    if (n < 2) return n;
5    #pragma omp task
6      x = fib(n-1);
7    #pragma omp task
8      y = fib(n-2);
9    #pragma omp taskwait
10   return (x+y);
11 }
```

Η αρχική κλήση στη `fib()` θα δημιουργήσει δύο εργασίες για τον υπολογισμό των x και y, κάθε μία εκ των οποίων θα καλέσει την ίδια συνάρτηση και θα δημιουργήσει δύο νέες εργασίες κ.ο.κ. Η γραμμή 9 είναι η απαραίτητη «πινελιά» για να δουλέψει το πρόγραμμα. Αν δεν υπάρχει, τότε τα δεδομένα στη γραμμή 10 δεν πρόκειται να έχουν τις σωστές τιμές, μιας και δεν γνωρίζουμε αν οι εργασίες θα εκτελεστούν αμέσως! Με την οδηγία `taskwait` εξασφαλίζουμε ότι οι δύο εργασίες που δημιουργούμε θα έχουν ολοκληρωθεί και τα x και y θα έχουν λάβει τις σωστές τιμές. Όμως, αυτό από μόνο του δεν είναι αρκετό. Έχουμε και εδώ πρόβλημα με τα δεδομένα. Από τη στιγμή που δεν υπάρχει παράλληλη περιοχή που να περικλείει λεκτικά τις δομές των εργασιών, ο μεταφραστής δεν μπορεί να αποδείξει αν κάποια μεταβλητή είναι κοινόχρηστη, και υποχρεωτικά θα τις κατηγοριοποιήσει όλες ως `firstprivate()`. Όμως, τότε οι μεταβλητές x και y θα είναι ιδιωτικές στην κάθε εργασία και άρα δεν θα πάρουμε σε αυτές τα επιθυμητά αποτελέσματα. Η λύση είναι να καθορίσουμε ρητά τις δύο αυτές μεταβλητές ως κοινόχρηστες. Μαζί με τον τελικό κώδικα στο Πρόγρ. 4.28, σημειώνουμε μία ακόμα φορά ότι είναι καλύτερο να καθορίζονται ρητά τα χαρακτηριστικά κοινοχρησίας όλων των μεταβλητών που συμμετέχουν στις εργασίες, ώστε να εξασφαλίζουμε πλήρως την ορθότητα του κώδικα.

Πριν ολοκληρώσουμε την ενότητα, πρέπει να παρατηρήσουμε δύο πράγματα σχετικά με το Πρόγρ. 4.28. Πρώτα από όλα, θα δημιουργηθούν πάρα πολλές εργασίες. Συγκεκρι-

```
int fib(int n) {
  int x, y;

  if (n < 2) return n;
  #pragma omp task shared(x) firstprivate(n) if(n > 20)
    x = fib(n-1);
  #pragma omp task shared(y) firstprivate(n) if(n > 20)
    y = fib(n-2);
  #pragma omp taskwait
  return (x+y);
}
```

Πρόγραμμα 4.29 *Αναδρομικός υπολογισμός αριθμών Fibonacci με εργασίες του OpenMP, δεύτερη έκδοση. (sas-omp-fib2.c)*

μένα, για τον 30ο αριθμό Fibonacci θα δημιουργηθούν 2,692,537 (!) εργασίες. Παρά το γεγονός ότι δεν υπάρχει κάποιος περιορισμός, όπως υπήρχε με το πλήθος νημάτων, οι εργασίες είναι υπερβολικά πολλές και σίγουρα θα υπάρχει χρονικό κόστος στη διαχείρισή τους. Κατά δεύτερον, η κάθε εργασία δεν κάνει σχεδόν τίποτε. Δημιουργεί 2 άλλες και κάνει μία πρόσθεση. Το κόστος δημιουργίας των εργασιών φαίνεται να είναι κατά πολύ μεγαλύτερο από τον χρόνο που απαιτείται για την εκτέλεσή τους. Για τους λόγους αυτούς, το OpenMP παρέχει μία φράση για τον περιορισμό των εργασιών που θα δημιουργηθούν, τη φράση `if(cond)`. Εφόσον η συνθήκη `cond` είναι αληθής, η εργασία θα δημιουργηθεί κανονικά, αλλιώς ο κώδικάς της θα εκτελεστεί *αμέσως* από το νήμα που συνάντησε τη δομή `task` (γλιτώνοντας έτσι τον χρόνο για τη δημιουργία της και την καταχώρησή της στο σύστημα για μελλοντική εκτέλεση). Η φράση `if()` πρέπει να σχεδιάζεται έτσι ώστε να επιτρέπει τη δημιουργία αρκετών εργασιών, προκειμένου να κρατούνται τα νήματα απασχολημένα, αλλά ταυτόχρονα όχι πολύ λεπτόκοκκων, ώστε κάθε εργασία να έχει αρκετή δουλειά και να δικαιολογεί τον χρόνο που χάνεται στη δημιουργία της και τη διαχείρισή της από το σύστημα. Το Πρόγρ. 4.29 είναι μια πολύ πιο αποδοτική έκδοση του Προγρ. 4.28. Εάν το `n` είναι πολύ μικρό, είναι καλύτερα να καταφύγουμε στον άμεσο (σειριακό) υπολογισμό από το να δημιουργήσουμε πληθώρα λεπτόκοκκων εργασιών.

4.11 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις

Με το κεφάλαιο αυτό μυηθήκαμε στον παράλληλο προγραμματισμό, μέσα από την ενασχόλησή μας με το μοντέλο κοινόχρηστου χώρου διευθύνσεων. Ένα παράλληλο πρόγραμμα στηρίζεται στην ύπαρξη μίας συλλογής από οντότητες εκτέλεσης που αναλαμβάνουν τους υπολογισμούς. Οι οντότητες εκτέλεσης είναι συνήθως νήματα, αν και μερικές φορές μπορεί

να είναι διεργασίες. Για το μοντέλο κοινόχρηστου χώρου διευθύνσεων, τα νήματα POSIX ή Pthreads αποτελούν την πιο ευρέως χρησιμοποιούμενη βιβλιοθήκη νημάτων. Όποιος και να είναι ο τύπος των οντοτήτων εκτέλεσης, πάντως, το μοντέλο υποθέτει ότι υπάρχει ένας μηχανισμός για τη δημιουργία τους αλλά και για την υποστήριξη κοινόχρηστων μεταβλητών μεταξύ τους.

Επιπλέον, θα πρέπει να παρέχονται δύο ακόμα ευκολίες, οι οποίες θεωρούνται απαραίτητες από τη στιγμή που υπάρχουν πολλαπλές οντότητες εκτέλεσης και κοινόχρηστες μεταβλητές: πρώτον, δομές για αμοιβαίο αποκλεισμό που πρέπει να χρησιμοποιούνται κατά την προσπέλαση κοινόχρηστων μεταβλητών και δεύτερον, δομές για συγχρονισμό, ούτως ώστε τα νήματα να ξεκινούν ή να τελειώνουν κάποιους υπολογισμούς τους όλα μαζί. Αμοιβαίος αποκλεισμός επιτυγχάνεται συνήθως με τη χρήση *κλειδαριών*, που στα Pthreads ονομάζονται mutexes. Η πιο συνηθισμένη μορφή συγχρονισμού είναι αυτή των *κλήσεων φραγής* ή barriers.

Τα νήματα POSIX ορίστηκαν επίσημα ως πρότυπο το 1995 [IEEE95]. Η αρχική διεπαφή των Pthreads δεν προέβλεπε κλήσεις φραγής. Στη συνέχεια, εκδόθηκε προαιρετική επέκταση του προτύπου, με επιπλέον κλήσεις για προχωρημένα νήματα πραγματικού χρόνου (advanced realtime threads), όπου συμπεριλαμβάνονταν και οι barriers [IEEE00]. Παρότι δεν είναι υποχρεωτικό να τα παρέχουν όλα τα συστήματα, εντούτοις είναι διαθέσιμα σχεδόν παντού. Μέχρι το πρόσφατο παρελθόν υπήρχε η απαίτηση να έχουν οριστεί κάποιες συγκεκριμένες σταθερές μέσα στο πρόγραμμα του χρήστη, ώστε να επιτραπεί η χρήση των barrier και συγκεκριμένα, θα έπρεπε να υπήρχε η εξής σταθερά:

```
#define _XOPEN_SOURCE 600
```

Πλέον, όμως, δεν υφίσταται κάποια τέτοια υποχρέωση και οι κλήσεις φραγής μπορούν να γίνουν από οποιοδήποτε πολυνηματικό πρόγραμμα. Ο προγραμματισμός με νήματα POSIX περιλαμβάνεται σχεδόν σε όλα τα συγγράμματα που ασχολούνται με τον παράλληλο προγραμματισμό και όχι μόνο (π.χ. σε βιβλία που ασχολούνται με τον προγραμματισμό συστημάτων, ακόμα και σε σειριακούς υπολογιστές, όπως το κλασικό βιβλίο του Stevens [StRa13]). Το βιβλίο του Butenhof [Bute97], όμως, ασχολείται μόνο με τα Pthreads και θεωρείται ως βασική αναφορά.

Πέρα από τις όποιες ευκολίες παρέχει κάποιο συγκεκριμένο σύστημα, είναι απαραίτητη η γνώση βασικών τεχνικών για τη δημιουργία αποδοτικών προγραμμάτων. Οι τεχνικές αυτές χρησιμοποιούνται κατά την ανάθεση δουλειάς στα νήματα και αποσκοπούν τόσο στην πλήρη εκμετάλλευση των πόρων (επεξεργαστών) όσο και στην ισοκατανομή του φόρτου για μέγιστη ταχύτητα. Ανάλογα με την περίσταση, οι τεχνικές της τμηματικής δρομολόγησης, της δρομολόγησης με άλματα και του διαχωρισμού σκακιέρας, όταν πρόκειται για παραλληλοποίηση απλών ή πολλαπλών βρόχων, μπορούν να οδηγήσουν σε βελτιστοποιημένα προγράμματα. Σε περιπτώσεις που οι επαναλήψεις των βρόχων είναι ανισοβαρείς, οι επεξεργαστές εκτελούν με διαφορετικές ταχύτητες ή το σύστημα δεν είναι αποκλειστικά αφιε-

ρωμένο (dedicated) στην εκτέλεση της δικής μας εφαρμογής, η αυτοδρομολόγηση αποτελεί μία άλλη στρατηγική δόμησης των προγραμμάτων, η οποία όπου μπορεί να εφαρμοστεί, έχει δυνατότητες να ισοκατανείμει αυτόματα τον φόρτο.

Το OpenMP ήρθε να αντικαταστήσει σε μεγάλο βαθμό τον απευθείας προγραμματισμό με νήματα, κρύβοντας αρκετές από τις λεπτομέρειες πίσω από πολύ απλές και εύχρηστες οδηγίες (directives). Η δημιουργία νημάτων (τα οποία δεν τα χειρίζεται ρητά ο προγραμματιστής), ο αμοιβαίος αποκλεισμός και ο συγχρονισμός καταλήγουν να είναι απλώς μία σημείωση / σχόλιο μέσα στον κώδικα. Με τη σωστή χρήση των οδηγιών αυτών, σε συνδυασμό με οδηγίες διαμοιρασμού εργασίας, είναι πολλές φορές εφικτό να παραλληλοποιηθεί ένα σειριακό πρόγραμμα με ελάχιστο κόπο. Γι' αυτό και το OpenMP αποτελεί, πλέον, τον δημοφιλέστερο τρόπο να προγραμματίζει κανείς στο μοντέλο κοινόχρηστου χώρου διευθύνσεων. Αν προσθέσει κανείς και τις υποδομές για *εργασιοκεντρικό παραλληλισμό* (task-based parallelism) με την οδηγία `task`, γίνεται αντιληπτό ότι πρόκειται για ένα μοντέλο με ευρύ πεδίο εφαρμογής. Τα βιβλία [CMDK01] και [CJP07] ασχολούνται αποκλειστικά με το OpenMP.

Το OpenMP ξεκίνησε ως ανοιχτό πρότυπο το 1997, για τη γλώσσα Fortran, και ένα χρόνο αργότερα για τις γλώσσες C/C++. Η έκδοση 2.0 δημοσιεύτηκε, αντίστοιχα, το 2000 και το 2002. Από την έκδοση 2.5 [OMP25] και μετά, το πρότυπο είναι ενιαίο για τις τρεις γλώσσες. Κάθε έκδοση ξεκαθάριζε κάποιες λεπτομέρειες και έκανε σχετικά περιορισμένες προσθήκες νέων φράσεων και κλήσεων βιβλιοθήκης στην προηγούμενη έκδοση. Η έκδοση 3.0 [OMP30] αποτέλεσε μεγάλη αλλαγή, καθώς βελτίωσε το πρότυπο από πολλές απόψεις, αλλά κυρίως εισήγαγε τις εργασίες (tasks). Η εμβέλεια του OpenMP ξέφυγε πλέον από τις εφαρμογές που βασίζονται μόνο σε βρόχους και επεκτάθηκε και σε αυτές που θεωρούνται ως ακανόνιστες. Βελτιώσεις στα σημεία έφερε και η επακόλουθη έκδοση 3.1 [OMP31], η οποία δημοσιεύτηκε το 2011, και η οποία αποτελεί την έκδοση που υποστηρίζουν οι περισσότεροι μεταφραστές OpenMP. Πληροφορίες για το OpenMP και τις εκδόσεις του μπορεί κανείς να βρει στην ιστοσελίδα του, `http://www.openmp.org`.

Η επιτυχία του OpenMP δεν ήρθε αμέσως. Μέχρι και την έκδοση 2.5 ήταν πολύ λίγοι οι μεταφραστές που το υποστήριζαν. Η κρίσιμη συγκυρία ήταν η εμφάνιση και ολοκληρωτική επικράτηση των πολυπύρηνων επεξεργαστών. Τότε μόνο έγινε αντιληπτή η ανάγκη να γραφτούν εφαρμογές παράλληλα, μιας και δεν υπήρχαν άλλα περιθώρια στη βελτίωση των επιδόσεων των μονοπύρηνων επεξεργαστών. Όμως, το να γραφτεί παράλληλο λογισμικό από προγραμματιστές που δεν είχαν προγραμματίσει παράλληλα στο παρελθόν, αποδείχτηκε εξαιρετικά δύσκολο και ασύμφορο. Αποτέλεσε και αποτελεί μεγαλύτερη πρόκληση από τη σχεδίαση ταχύτερου υλικού [SuLa05]. Το OpenMP ήρθε να δώσει μία από τις καλύτερες λύσεις, λόγω των ευκολιών που παρέχει ακόμα και σε προγραμματιστές που δεν έχουν μυηθεί στον παραλληλισμό. Το OpenMP βρίσκεται πλέον στην έκδοση 4.0 [OMP40], η οποία έχει φέρει σημαντικές προσθήκες στο πρότυπο, σε μία προσπάθεια να επεκτείνει την εφαρμογή του και στον χώρο των επιταχυντών και των μονάδων γραφικής επεξεργασίας.

Ακόμα, όμως, δεν υπάρχει ικανή υποστήριξη από τους γνωστούς μεταφραστές.

Ας μην ξεχνάμε όμως, ότι το OpenMP δεν είναι πανάκεια. Ενώ είναι απλό να αντλήσει κάποιος επιδόσεις χωρίς μεγάλο κόπο, εντούτοις το να καταφέρει οι επιδόσεις αυτές να είναι οι μεγαλύτερες δυνατές δεν είναι πάντα εύκολο. Και αυτό διότι δεν επιτρέπει λεπτομερή διαχείριση των νημάτων αλλά και της τοποθέτησης των δεδομένων. Κρύβει, ουσιαστικά, τόσο τα νήματα όσο και τις λεπτομέρειες της μηχανής στην οποία εκτελείται το πρόγραμμα με ό,τι καλό και ό,τι κακό συνεπάγεται αυτό.

Κλείνοντας, πρέπει να σημειώσουμε ότι ως μοντέλο, ο προγραμματισμός με κοινόχρηστο χώρο διευθύνσεων, περιλαμβάνεται σχεδόν σε όλα τα βιβλία που ασχολούνται με τον παράλληλο προγραμματισμό (για παράδειγμα, [Quin03, WiAl04, LiSn08, Pach11]). Στην ελληνική βιβλιογραφία, οι Πάντζιου, Μάμαλης και Τομάρας έχουν συμπεριλάβει και τον προγραμματισμό με το OpenMP στο βιβλίο τους [PMT13]. Πέρα από όλα αυτά, αξίζει να ξεχωρίσουμε τα βιβλία των [MSM04] και [MRR12], διότι όχι μόνο περιλαμβάνουν το μοντέλο κοινόχρηστου χώρου διευθύνσεων, αλλά ασχολούνται με τον γενικότερο τρόπο σκέψης του παράλληλου προγραμματισμού, με λεπτομερή ανάλυση προγραμματιστικών υποδειγμάτων (patterns) που κάνουν την εμφάνισή τους σε μεγάλη ποικιλία εφαρμογών. Τέλος, τόσο για τα νήματα, όσο και για το OpenMP υπάρχει τεράστια συλλογή από βοηθήματα και παραδείγματα στο διαδίκτυο, που μπορούν να συμπληρώσουν τη βιβλιογραφία και να δώσουν αφορμή για περαιτέρω εμβάθυνση.

Προβλήματα

4.1 – Μπορείτε να δείτε κάποιο πρόβλημα αν όλα τα νήματα εκτελούσαν τον εξής κώδικα, όπου το `mylock` είναι ένα mutex:

```
while (1)
{
  pthread_mutex_lock(&mylock);
  if (i < 100)
    sum += i;     /* sum is shared, i is private*/
  else
    break;
  pthread_mutex_lock(&mylock);
  i++;
}
```

4.2 – Σε μεγαλύτερα προγράμματα υπάρχουν αρκετές κρίσιμες περιοχές διάσπαρτες στον

κώδικα κάθε διεργασίας. Τι θα λέγατε αν κάποιος σας πρότεινε το εξής, προκειμένου να μειωθεί το μέγεθος του κώδικα: αντί να χρησιμοποιείτε πολλά ζεύγη κλείδωμα-ξεκλείδωμα (ένα για κάθε κρίσιμη περιοχή στον κώδικα), να χρησιμοποιήσετε ένα μόνο κλείδωμα πριν την πρώτη κρίσιμη περιοχή και ένα ξεκλείδωμα ακριβώς μετά την τελευταία.

4.3 – Μπορείτε να εξηγήσετε γιατί η τυπική χρήση των μεταβλητών συνθήκης πρέπει να είναι όπως στο Πρόγρ. 4.4; Συγκεκριμένα:

- Γιατί χρειάζεται να κλειδώσει την κλειδαριά και το νήμα Β; (*Βοήθεια*: θυμηθείτε ότι τα σήματα που δεν παραλαμβάνονται, χάνονται.)
- Γιατί το νήμα Α χρησιμοποιεί `while` και όχι ένα απλό `if`; (*Βοήθεια*: η εκτέλεση των νημάτων είναι χρονικά απρόβλεπτη και δεν γνωρίζουμε τι κάνει παρακάτω το νήμα Β.)

4.4 – Προσπαθήστε να υλοποιήσετε έναν δικό σας barrier για νήματα, κάνοντας χρήση ενός μετρητή και μιας μεταβλητής συνθήκης. Διατηρήστε παρόμοια διεπαφή με τα Pthreads υλοποιώντας μία συνάρτηση αρχικοποίησης `bar_init()` και μια συνάρτηση αναμονής `bar_wait()`. Η λειτουργία του θα πρέπει να είναι η εξής:

- Κατά την αρχικοποίηση θα πρέπει να ορίζεται το πλήθος (N) των νημάτων που θα συμμετέχουν.
- Κατά την κλήση αναμονής, ο μετρητής θα πρέπει να μετρά πόσα νήματα έχουν εισέλθει στον barrier. Εάν το πλήθος τους είναι λιγότερο από $N-1$, το νήμα που κάνει την κλήση θα πρέπει να μπλοκάρει. Εάν είναι ακριβώς $N-1$, το νήμα αυτό θα πρέπει να ξυπνά όλα τα άλλα.

Αφού το υλοποιήσετε, προσέξτε την εξής λεπτομέρεια: τι θα γίνει αν στο πρόγραμμα του χρήστη υπάρχουν δύο σημεία συγχρονισμού, το ένα μετά το άλλο; Θα λειτουργεί πάντα σωστά ο barrier που φτιάξατε αν κληθεί δεύτερη φορά; Αν όχι, κάντε τις απαραίτητες διορθώσεις.

4.5 – Δρομολογώντας τις επαναλήψεις ενός βρόχου, αν $i_{\min}$ είναι το μικρότερο και $i_{\max}$ είναι το μεγαλύτερο πλήθος επαναλήψεων που δίνονται σε νήμα, η διαφορά $\rho = i_{\max} - i_{\min}$ είναι ένα μέτρο ανισορροπίας φόρτου (imbalance) ανάμεσα στα νήματα. Εφόσον $\rho \leq 1$, ο φόρτος είναι πλήρως ισορροπημένος και οι επαναλήψεις είναι κατανεμημένες όσο πιο ομοιόμορφα γίνεται. Θεωρήστε ένα πρόγραμμα με m νήματα που κάνει χρήση της δρομολόγησης με άλματα, για ένα βρόχο με N επαναλήψεις. Τι μπορείτε να πείτε για την τιμή του ρ στην περίπτωση αυτή;

4.6 – Δώστε πρόγραμμα πολλαπλασιασμού πίνακα επί διάνυσμα, χρησιμοποιώντας τμηματική δρομολόγηση στο βρόχο `i`, και `NPROC` $< N$ νήματα. Υποθέστε τις παρακάτω τιμές:

```
#define N     512     /* Dimension */
#define NPROC 32      /* # cores / threads */
#define WORK  N/NPROC /* # result elements per thread */
```

4.7 – Επαναλάβετε το Πρόβλημα 4.6, αλλά με χρήση δρομολόγησης με άλματα.

4.8 – Υλοποιήστε τον πολλαπλασιασμό πινάκων, χρησιμοποιώντας τμηματική δρομολόγηση στον βρόχο `i` (βλ. Ενότητα 4.7.2), υποθέτοντας `NPROC` $< N$ νήματα.

4.9 – Υλοποιήστε τον πολλαπλασιασμό τετραγωνικών πινάκων όπως στο Πρόγρ. 4.18 στη γενική περίπτωση. Δηλαδή, θεωρήστε ότι `NTHR` $< M^2$, και επομένως, το κάθε νήμα μπορεί να υπολογίζει παραπάνω από έναν υποπίνακα του αποτελέσματος.

4.10 – Υλοποιήστε τον πολλαπλασιασμό πινάκων, χρησιμοποιώντας νήματα και αυτοδρομολόγηση. Θεωρήστε ως στοιχειώδη εργασία τον υπολογισμό ενός υποπίνακα του αποτελέσματος. Καθορίστε το μέγεθος S των υποπινάκων μόνοι σας (αρκεί να διαιρεί το συνολικό μέγεθος των πινάκων, N), οπότε θα προκύψουν συνολικά $(N/S)^2$ εργασίες.

5 Προγραμματισμός με Μεταβίβαση Μηνυμάτων

Σε αυτό το κεφάλαιο του βιβλίου θα ασχοληθούμε με το προγραμματιστικό μοντέλο μεταβίβασης μηνυμάτων (message passing model), κατά πολλούς το πιο ευρέως χρησιμοποιούμενο μοντέλο στον χώρο των υπολογισμών υψηλών επιδόσεων (high performance computing, HPC).

Θα γνωρίσουμε τις βασικές αρχές του μέσα από τις προγραμματιστικές δομές που παρέχονται συνήθως (οι οποίες είναι πολλές φορές μόνο δύο!), αλλά και από απλές τεχνικές που θα προκύψουν με το σχεδιασμό μικρών εφαρμογών. Στη συνέχεια, θα δούμε και θα εφαρμόσουμε πιο προχωρημένες τεχνικές που προκύπτουν κυρίως από την ανάγκη αύξησης των επιδόσεων.

Η μελέτη μας αυτή θα γίνει κάνοντας χρήση του πιο διαδεδομένου προτύπου για τον προγραμματισμό με μεταβίβαση μηνυμάτων: του Message Passing Interface ή απλά MPI.

Το μοντέλο μεταβίβασης μηνυμάτων θεωρείται ένα από τα σχετικά δύσκολα μοντέλα παράλληλου προγραμματισμού καθώς, όπως θα δούμε, ο προγραμματιστής θα πρέπει να καθορίζει επακριβώς τα σημεία στα οποία σταματούν, προσωρινά, οι υπολογισμοί και αρχίζουν οι επικοινωνίες. Μάλιστα, θα πρέπει να καθορίζει πλήρως το περιεχόμενο αλλά και τους αποστολείς και αποδέκτες των μεταφερόμενων μηνυμάτων. Και όλα αυτά διότι, πέρα από τη ρητή διαμοίραση των υπολογιστικών εργασιών, ο προγραμματιστής πρέπει να κάνει επιπλέον ρητή διαμοίραση / τοποθέτηση των δεδομένων του προγράμματος. Όμως, αποτελούσε (και αποτελεί) το ιδανικό μοντέλο για «φθηνό» παράλληλο υπολογισμό.

Η μεταβίβαση μηνυμάτων αποτελεί τον βασικό τρόπο προγραμματισμού των διαφόρων εμπορικών πολυεπεξεργαστών κατανεμημένης μνήμης. Ιστορικά, κάθε κατασκευαστής συστημάτων αυτού του τύπου παρείχε και κάποιον δικό του μεταφραστή για τις γλώσσες C ή/και Fortran, στις οποίες υπήρχαν οι κατάλληλες επεκτάσεις για τον προγραμματισμό με μεταβίβαση μηνυμάτων. Οι διαφορές μεταξύ των αντίστοιχων εκδόσεων των γλωσσών αυτών μπορεί να μην ήταν ιδιαίτερα μεγάλες από κατασκευαστή σε κατασκευαστή, ήταν όμως αρκετές για να υπάρχει πλήρης ασυμβατότητα μεταξύ των προγραμμάτων. Κάθε πρόγραμμα ήταν, σε επίπεδο πηγαίου κώδικα, άμεσα συνυφασμένο με τον υπολογιστή που θα το εκτελούσε, και απαιτούσε προσπάθεια για να μεταφερθεί σε άλλο υπολογιστή. Το γεγονός αυτό, συνδυαζόμενο με την ανάδειξη των τοπικών δικτύων ως εναλλακτικών, οικονομικών παράλληλων υπολογιστών, οδήγησε στην υιοθέτηση καθολικά αποδεκτών προτύπων, με πιο σημαντικό το MPI (Message Passing Interface). Το MPI αποτελεί πλέον μονόδρομο για προγραμματισμό με μεταβίβαση μηνυμάτων τόσο σε καθαυτό παράλληλους υπολογιστές κατανεμημένης μνήμης, όσο και σε τοπικές συστάδες αποτελούμενες από σταθμούς εργασίας.

Η ανάπτυξη του μοντέλου μεταβίβασης μηνυμάτων είναι συνυφασμένη με την άνοδο και επικράτηση του λεγόμενου παράλληλου υπολογιστή του «φτωχού»: το πανταχού παρών και ανέξοδο τοπικό δίκτυο από ανεξάρτητους υπολογιστές. Οι σταθμοί εργασίας, ενωμένοι σε ένα τοπικό δίκτυο (γνωστό και ως συστάδα), αποτελούν έναν «πολυϋπολογιστή» με εγγενώς κατανεμημένη μνήμη που απλά δεν διαθέτει πάρα πολύ μεγάλες ταχύτητες επικοινωνίας μεταξύ των υπολογιστών-κόμβων, όπως είδαμε και στην Ενότητα 3.6.1. Σε ένα τέτοιο περιβάλλον, το μοντέλο μεταβίβασης μηνυμάτων κάνει την εμφάνισή του αβίαστα—ειδικά αν υπολογίσει κανείς την εμπειρία που υπάρχει σε προγραμματισμό κατανεμημένων συστημάτων με τη βιβλιοθήκη των υποδοχών (sockets) του UNIX. Ο προγραμματισμός με αποστολή και λήψη μηνυμάτων μέσω sockets θα μπορούσε να αποτελεί ένα καλό παράδειγμα προγραμματισμού στο μοντέλο που μελετάμε σε αυτό το κεφάλαιο. Στην πράξη, όμως, για παράλληλο προγραμματισμό σε τοπικό δίκτυο δεν χρησιμοποιούνται άμεσα τα sockets, αλλά βιβλιοθήκες όπως το MPI, οι οποίες είναι συνήθως χτισμένες πάνω από τα sockets.

Βλέπουμε, λοιπόν, ότι το μοντέλο μεταβίβασης μηνυμάτων, εκτός από κατάλληλο για μηχανές MIMD με κατανεμημένη μνήμη, είναι επίσης εφαρμόσιμο για παράλληλο προγραμ-

ματισμό που εκμεταλλεύεται την υπολογιστική ισχύ που υπάρχει σε ένα τοπικό δίκτυο, κάτι που είναι γνωστό και ως *δικτυακός παράλληλος προγραμματισμός* ή *προγραμματισμός υπολογιστικών συστάδων* (cluster computing). Και μάλιστα, με περιβάλλοντα όπως το παλαιότερο Parallel Virtual Machine (PVM), το δίκτυο μπορεί να είναι ακόμα και ανομοιογενές, δηλαδή να αποτελείται από διαφορετικού τύπου υπολογιστές.

Το μοντέλο μεταβίβασης μηνυμάτων είναι το δεύτερο είδος παραλληλισμού ελέγχου, μετά το μοντέλο κοινού χώρου διευθύνσεων. Προϋποθέτει και αυτό την ύπαρξη πολλαπλών διεργασιών (processes) που εργάζονται ταυτόχρονα. Όμως, σε αντίθεση με το μοντέλο κοινού χώρου διευθύνσεων, *οι διεργασίες δεν μπορούν να έχουν καμία μεταβλητή κοινή*.

Από τη στιγμή που οι διεργασίες δεν έχουν τίποτε κοινό μεταξύ τους, δεν υπάρχει, επίσης, καμία ανάγκη για αμοιβαίο αποκλεισμό. Το μόνο, τελικά, που είναι απαραίτητο για την επικοινωνία και συνεργασία είναι δομές για αποστολή και λήψη μηνυμάτων μεταξύ των διεργασιών. Για αυτόν τον λόγο, το μοντέλο μεταβίβασης, στην βασική μορφή του, είναι *πολύ απλό στην περιγραφή αλλά και στην υλοποίησή του·* τις δύο, μόλις, βασικές δομές του θα τις δούμε στην Ενότητα 5.1, ενώ η Ενότητα 5.2 εμβαθύνει λίγο περισσότερο στον τρόπο που λειτουργούν οι δομές αυτές.

Στην Ενότητα 5.3, θα δούμε μερικές απλές εφαρμογές που θα μας μυήσουν στον τρόπο προγραμματισμού με μεταβίβαση μηνυμάτων. Στις Ενότητες 5.4 και 5.5, θα δούμε κάποιες προχωρημένες δομές και μορφές επικοινωνίας, οι οποίες μπορούν να βελτιστοποιήσουν κατά πολύ τα προγράμματά μας. Κάποιες λιγότερο συνηθισμένες ευκολίες και δυνατότητες του MPI περιγράφονται στην Ενότητα 5.6. Τέλος, συνοψίζουμε το κεφάλαιο στην Ενότητα 5.7 όπου, επίσης, κάνουμε μία μικρή ιστορική αναδρομή στην εξέλιξη του MPI.

5.1 Βασικές δομές

Στην ενότητα αυτή, θα δούμε τις βασικές δομές που παρέχονται σε κάποιον, προκειμένου να προγραμματίσει στο μοντέλο μεταβίβασης μηνυμάτων. Αν εξαιρέσουμε κάποιες ευκολίες για τη διαχείριση διεργασιών (οι οποίες δεν είναι πάντα αναγκαίες, ούτε και προσφέρονται από όλα τα συστήματα), θα δούμε ότι απαραίτητες είναι μόνο δύο κύριες κλήσεις, αυτές για την αποστολή και τη λήψη μηνυμάτων. Ταυτόχρονα, θα δείχνουμε και τη μορφή των δομών αυτών στο MPI, μιας και πρακτικά αποτελεί τη μόνη επιλογή που έχει κάποιος για προγραμματισμό στο μοντέλο αυτό.

5.1.1 Διεργασίες

Ένα πρόγραμμα παραλληλισμού ελέγχου είτε είναι κοινού χώρου διευθύνσεων είτε είναι μεταβίβασης μηνυμάτων, υλοποιείται από ένα σύνολο οντοτήτων εκτέλεσης, όπως νήματα ή

διεργασίες. Στην περίπτωση της μεταβίβασης μηνυμάτων, ειδικά αν αναλογιστεί κανείς ότι στα συστήματα κατανεμημένης μνήμης εμπλέκονται πολλές φορές και ανεξάρτητοι υπολογιστές, οι οντότητες εκτέλεσης είναι σχεδόν πάντα *διεργασίες*. Όπως είδαμε και στο Κεφάλαιο 4, θα πρέπει, επομένως, κάθε σύστημα προγραμματισμού για το μοντέλο μας να διαθέτει μηχανισμούς για τη δημιουργία και διαχείριση των διεργασιών.

Σε συγκεκριμένους πολυεπεξεργαστές κατανεμημένης μνήμης, όπως π.χ. στον παλαιότερο iPSC/860 της Intel, η δημιουργία διεργασιών γινόταν με την κλασική `fork()` του UNIX που είδαμε στο Κεφάλαιο 4, η οποία δημιουργούσε ένα νέο αντίτυπο της καλούσας διεργασίας και το έστελνε προς εκτέλεση σε κάποιον ανενεργό επεξεργαστή. Αντίθετα, στον υπολογιστή nCUBE, οι διεργασίες ήταν προκαθορισμένες και δεν μπορούσαν να δημιουργηθούν δυναμικά (δηλαδή κατά τη διάρκεια της εκτέλεσης). Συγκεκριμένα, ο προγραμματιστής έγραφε ένα πρόγραμμα και πριν την έναρξη της εκτέλεσής του, το λειτουργικό σύστημα δημιουργούσε αντίτυπα του προγράμματος, προκειμένου να τα εκτελέσουν όλοι οι κόμβοι. Αυτό είναι το λεγόμενο μοντέλο SPMD (single-program multiple-data) που συζητήσαμε στο Ενότητα 1.4.1· όλοι οι επεξεργαστές εκτελούν αντίγραφα του ίδιου κώδικα, σε δικά του δεδομένα ο καθένας. Αυτός είναι και ο πιο συνηθισμένος τρόπος προγραμματισμού στο μοντέλο μεταβίβασης μηνυμάτων.

Σε γενικές γραμμές, πάντως, μπορεί να υποθέσει κανείς με ασφάλεια ότι σε πολλά συστήματα δεν παρέχονται ευκολίες για δυναμική δημιουργία και διαχείριση των διεργασιών—απλά υπάρχει κάποιος μηχανισμός με τον οποίο δημιουργούνται αρχικά όσες διεργασίες απαιτούνται, ως αντίγραφα κάποιας δεδομένης διεργασίας. Δεν θα ασχοληθούμε, επομένως, και εμείς με τις ιδιαιτερότητες της δημιουργίας διεργασιών. Θα υποθέσουμε ότι όλες οι διεργασίες είναι ίδιες (SPMD) και με κάποιον τρόπο δημιουργούνται στην αρχή της εκτέλεσης. Το MPI, το οποίο θα χρησιμοποιήσουμε εκτενώς στο κεφάλαιο αυτό, λειτουργεί με αυτόν τον τρόπο. Αρχικά, δεν προέβλεπε κλήσεις δημιουργίας και χειρισμού διεργασιών. Οι διάφορες υλοποιήσεις του αντιγράφουν το ίδιο πρόγραμμα στους κόμβους του συστήματος και το εκτελούν. Η εφαρμογή δημιουργείται σαν να πρόκειται για μία και μόνο διεργασία. Αφού μεταφραστεί, μπορεί κανείς να την εκτελέσει με όσες διεργασίες επιθυμεί· η εκτέλεση ενός προγράμματος `a.out` με `N` διεργασίες, γίνεται από τη γραμμή εντολών ως εξής:

```
% mpirun -np N a.out
```

όπου το `mpirun` είναι μία βοηθητική εφαρμογή του MPI που αναλαμβάνει να ενεργοποιήσει τους κόμβους του συστήματος, να δημιουργήσει `N` διεργασίες-αντίτυπα του προγράμματος `a.out` και να εκκινήσει την εκτέλεσή τους. Στις πιο πρόσφατες εκδόσεις του MPI, όμως, υπάρχουν επιπλέον κάποιες στοιχειώδεις λειτουργίες για δυναμική δημιουργία διεργασιών, τις οποίες θα δούμε αργότερα.

Από τη στιγμή που θα δημιουργηθούν με τον έναν ή τον άλλο τρόπο οι διεργασίες, αυτό που είναι απαραίτητο είναι η δυνατότητα να *διαφοροποιηθούν* μεταξύ τους, ώστε να

εκτελέσει η κάθε μία αυτό που της αντιστοιχεί. Επομένως, όλα τα συστήματα παρέχουν μία κλήση, η οποία επιστρέφει την *ταυτότητα* της διεργασίας που την καλεί. Επιπλέον, υπάρχει πάντα και μία κλήση που επιστρέφει το συνολικό αριθμό των διεργασιών που συμμετέχουν στην εκτέλεση. Στο MPI οι δύο κλήσεις είναι:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

όπου στις ακέραιες μεταβλητές `myid` και `nproc` επιστρέφονται, αντίστοιχα, η ταυτότητα της καλούσας διεργασίας (το MPI την ονομάζει *τάξη*–rank) και το συνολικό πλήθος των διεργασιών. Προσέξτε ότι:

- το `nproc` θα έχει την τιμή που έδωσε ο χρήστης κατά την εκτέλεση του προγράμματος με το `mpirun`, και επομένως, κατά κάποιο τρόπο, το πλήθος των διεργασιών δεν είναι πάντα προβλέψιμο από τον προγραμματιστή,
- οι ταυτότητες των διεργασιών αριθμούνται ακολουθιακά από 0 ως `nproc` $-$ 1.

Το `MPI_COMM_WORLD` είναι μία παράμετρος που μπαίνει σε κάθε κλήση του MPI και ονομάζεται *εξ ορισμού communicator*. Περισσότερα για αυτή την παράμετρο θα πούμε αργότερα.

5.1.2 Αποστολή και λήψη μηνυμάτων

Δεδομένων κάποιων διεργασιών, το μοντέλο μεταβίβασης μηνυμάτων στηρίζεται κυρίως σε δύο κλήσεις: μία για την *αποστολή* και μία για την *λήψη* μηνυμάτων. Όσον αφορά στην αποστολή μηνύματος, η διεργασία-αποστολέας θα πρέπει να προσδιορίσει, κατ' ελάχιστον:

- το χώρο στη μνήμη που βρίσκεται το μήνυμα που πρέπει να αποσταλεί,
- το μέγεθος του μηνύματος, και
- την ταυτότητα της διεργασίας-παραλήπτη.[1]

Όμως, σχεδόν όλα τα συστήματα προσφέρουν επιπλέον ευκολίες στον προγραμματιστή. Για παράδειγμα, το μήνυμα μπορεί να έχει και μία *ετικέτα* (tag), η οποία είναι συνήθως ένα απλός ακέραιος αριθμός, και είναι στη διάθεση του προγραμματιστή να δώσει σε αυτήν ό,τι τιμή θέλει.

Η αποστολή μηνυμάτων στο MPI περιέχει όλα τα παραπάνω χαρακτηριστικά και γίνεται με μία κλήση της μορφής:

```
MPI_Send(buf, n, dtype, torank, tag, MPI_COMM_WORLD);
```

όπου:

[1] Τη διεργασία-αποστολέα θα τη λέμε και *πηγή* (source), ενώ τη διεργασία-παραλήπτη θα τη λέμε και *προορισμό* (destination) του μηνύματος.

- buf είναι η διεύθυνση (δείκτης) στη μνήμη που βρίσκεται αποθηκευμένο το μήνυμα,
- n είναι *το πλήθος των στοιχείων* του μηνύματος,
- dtype είναι *ο τύπος των στοιχείων* του μηνύματος,
- torank είναι η ταυτότητα της διεργασίας που θα λάβει το μήνυμα,
- tag είναι ένας οποιοσδήποτε ακέραιος αριθμός που χρησιμεύει ως ετικέτα.

Αυτό που πρέπει να γίνει κατανοητό είναι ότι για το MPI κάθε μήνυμα είναι *ένας πίνακας από στοιχεία*. Επομένως, ο προγραμματιστής πρέπει να προσδιορίσει πού ξενικά ο πίνακας αυτός στην μνήμη (buf), πόσα στοιχεία έχει (n), καθώς και τον τύπο των στοιχείων αυτών (dtype). Ο τύπος των στοιχείων μπορεί να είναι σχεδόν οτιδήποτε, ακόμα και δομές (structs) που έχουν οριστεί από τον προγραμματιστή. Πρακτικά, οι πιο συνηθισμένοι τύποι είναι χαρακτήρες, ακέραιοι, κλπ., οι οποίοι προσδιορίζονται με τις εξής σταθερές του MPI: MPI_CHAR, MPI_INT, MPI_SHORT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE κ.ο.κ.

Έτσι, για παράδειγμα, αν επιθυμούμε την αποστολή ενός ακέραιου αριθμού στη διεργασία 8, θα πρέπει να τον προσδιορίσουμε ως πίνακα με 1 στοιχείο, τύπου MPI_INT, όπως στον παρακάτω κώδικα:

```
int    x = 3;
double arr[100] = {1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 1.7, 1.8, 1.9};

MPI_Send(&x, 1, MPI_INT, 8, 0, MPI_COMM_WORLD);
MPI_Send(arr, 5, MPI_DOUBLE, 8, 0, MPI_COMM_WORLD);
```

Στον παραπάνω κώδικα στέλνουμε και δεύτερο μήνυμα στη διεργασία 8, όπου περιλαμβάνουμε τα 5 πρώτα στοιχεία του πίνακα arr. Και στις δύο περιπτώσεις, βάζουμε τυπικά ως ετικέτα το 0, χωρίς να την χρησιμοποιούμε για κάτι.

Η αποστολή μηνύματος σε μία διεργασία δεν συνεπάγεται και την παραλαβή του από αυτήν· οι διεργασίες παραλαμβάνουν μηνύματα όποτε και από όποιον επιθυμούν, και μόνο αν το δηλώσουν ρητά. Υπάρχει, επομένως, κλήση για την *λήψη / παραλαβή* μηνυμάτων, την οποία θα πρέπει να κάνει όποια διεργασία περιμένει κάτι από κάποια άλλη. Η κλήση έχει παραμέτρους αντίστοιχες με αυτές της αποστολής, και στο MPI γίνεται ως εξής:

```
MPI_Recv(buf, n, dtype, fromrank, tag, MPI_COMM_WORLD, statusptr);
```

όπου, πλέον:

- buf είναι η διεύθυνση (δείκτης) στη μνήμη όπου θα αποθηκευτεί το μήνυμα όταν ληφθεί. Προφανώς, εκεί θα πρέπει να υπάρχει ο απαιτούμενος χώρος.
- n είναι το πλήθος των στοιχείων του μηνύματος
- dtype είναι ο τύπος των στοιχείων του μηνύματος

- fromrank είναι η ταυτότητα της διεργασίας η οποία στέλνει το μήνυμα
- tag είναι η ετικέτα
- statusptr είναι μία παράμετρος η οποία χρησιμεύει για να αποκομίσουμε πληροφορίες για το ληφθέν μήνυμα. Αν δεν χρειαζόμαστε τις πληροφορίες αυτές, μπορούμε να της δώσουμε την τιμή MPI_STATUS_IGNORE ώστε να ενημερώσουμε το MPI ότι δεν μας ενδιαφέρει.

Προσέξτε ότι, κατά την κλήση της λήψης, η διεργασία καθορίζει ακριβώς τον αποστολέα αλλά και την ετικέτα του μηνύματος που περιμένει να λάβει. Αν καταφθάσει μήνυμα με διαφορετική ετικέτα ή από άλλη πηγή, *δεν θα παραληφθεί*. Το μήνυμα θα μείνει κάπου αποθηκευμένο (το MPI φροντίζει για αυτό) και η παραλαβή θα γίνει μόνο εφόσον η διεργασία, αργότερα, καλέσει πάλι την MPI_Recv() με τις κατάλληλες παραμέτρους. Η διεργασία 8, μπορεί να λάβει τα μηνύματα που στείλαμε παραπάνω, με τον εξής κώδικα:

```
int    y;
double num[5];

MPI_Recv(&y, 1, MPI_INT, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(num, 5, MPI_DOUBLE, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

όπου έχουμε υποθέσει ότι αποστολέας είναι η διεργασία 3. Προσέξτε ότι, η ετικέτα πρέπει να έχει την τιμή 0, μιας και αυτήν χρησιμοποίησε η πηγή κατά την αποστολή. Επίσης, προσέξτε ότι, το MPI εξασφαλίζει πως *τα μηνύματα που στάλθηκαν από την ίδια πηγή και έχουν την ίδια ετικέτα, θα παραληφθούν στον προορισμό με τη σειρά που στάλθηκαν*. Έτσι, δεν υπάρχει περίπτωση να παραληφθεί πρώτα το δεύτερο και μετά το πρώτο μήνυμα από αυτά που στάλθηκαν παραπάνω.

Είναι πολύ συνηθισμένη, όμως, η περίπτωση όπου ένας παραλήπτης περιμένει να λάβει *οποιοδήποτε* μήνυμα του αποσταλεί, δηλαδή από οποιαδήποτε πηγή ή / και με οποιαδήποτε ετικέτα. Η διεργασία αυτή δεν μπορεί, επομένως, να προσδιορίσει τον αποστολέα και την ετικέτα κατά την κλήση της λήψης. Σε όλα τα συστήματα υπάρχει πρόβλεψη για τέτοιες περιπτώσεις, και ο προγραμματιστής μπορεί να περνά ειδικές τιμές στις αντίστοιχες παραμέτρους. Στο MPI, στην τέταρτη και πέμπτη παράμετρο της MPI_Recv(), μπορεί κανείς να περάσει τις τιμές MPI_ANY_SOURCE και MPI_ANY_TAG, αντίστοιχα. Έτσι για παράδειγμα, η κλήση

```
MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

θα έχει ως αποτέλεσμα την παραλαβή οποιουδήποτε ακεραίου έχει καταφθάσει. Εδώ βρίσκεται και η χρησιμότητα της τελευταίας παραμέτρου: αφού παραλάβουμε το μήνυμα, από αυτή μπορούμε να βρούμε και ποια ήταν η πηγή και τι ετικέτα είχε. Η παράμετρος statusptr δείχνει σε μία δομή η οποία διαθέτει, μεταξύ άλλων, δύο πεδία MPI_SOURCE και

MPI_TAG, τα οποία περιέχουν την ταυτότητα του αποστολέα και την ετικέτα μηνύματος, αντίστοιχα. Ο τρόπος χρήσης της φαίνεται στον κώδικα που ακολουθεί:

```
int        y;
MPI_Status status;   /* Must be of type MPI_Status */

MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
         &status);
printf("Message received from process %d, with tag %d.\n",
        status.MPI_SOURCE, status.MPI_TAG);
```

Όσο και αν φαίνεται περίεργο, οι δύο αυτές κλήσεις για αποστολή και λήψη είναι αρκετές για να προγραμματίσει κανείς στο μοντέλο μεταβίβασης μηνυμάτων αν και, όπως θα δούμε αργότερα, μπορεί να παρέχονται επιπλέον δομές, με σκοπό αφενός μεν την προγραμματιστική διευκόλυνση, αφετέρου δε τη βελτίωση των επιδόσεων. Με βάση αυτές τις δύο κλήσεις, θα δούμε στη συνέχεια τις κύριες προγραμματιστικές τεχνικές που επιστρατεύονται στη μεταβίβαση μηνυμάτων, αφού πρώτα, όμως, δούμε με λίγο περισσότερη λεπτομέρεια την έννοια της επικοινωνίας.

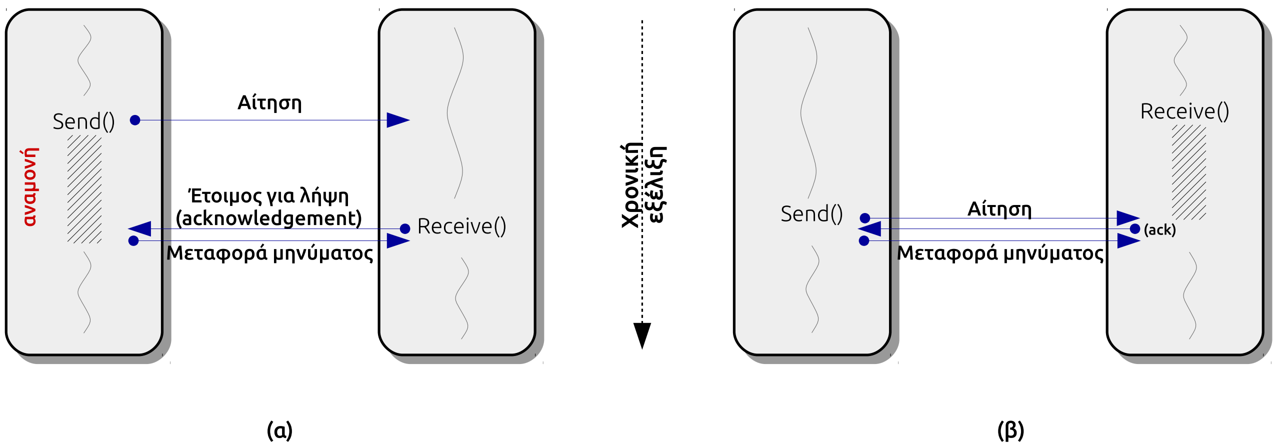
5.2 Η λειτουργία της επικοινωνίας

Πριν προχωρήσουμε στην εφαρμογή των κλήσεων που μάθαμε για την υλοποίηση προγραμμάτων με μεταβίβαση μηνυμάτων, είναι απαραίτητο να εμβαθύνουμε λίγο στη βασική λειτουργία της επικοινωνίας μεταξύ διεργασιών, κάτι που θα μας δώσει τα εργαλεία να κατανοήσουμε και να εξηγήσουμε κάποιες φαινομενικά «περίεργες» καταστάσεις που θα συναντήσουμε αργότερα.

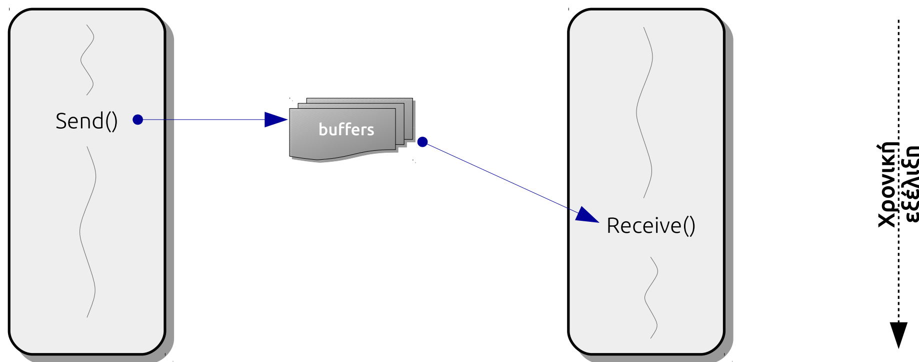
Όπως είδαμε, η κλήση αποστολής ενός μηνύματος από την πηγή πρέπει να συνοδεύεται και από την αντίστοιχη κλήση λήψη μηνύματος από τον προορισμό, προκειμένου να ολοκληρωθεί η επικοινωνία μεταξύ των δύο διεργασιών. Όμως, οι δύο διεργασίες λειτουργούν ανεξάρτητα η μία από την άλλη, και οι σχετικές τους ταχύτητες δεν είναι προβλέψιμες. Δημιουργούνται, έτσι, εύλογα ερωτήματα όπως: τι γίνεται στην περίπτωση που οι κλήσεις αποστολής και λήψης δεν γίνονται την ίδια χρονική στιγμή; Τι γίνεται αν ο παραλήπτης κάνει κλήση λήψης πριν καν ο αποστολέας εκτελέσει την αποστολή; Η απάντηση στα ερωτήματα αυτά δίνεται από το *μοντέλο επικοινωνίας* που υλοποιεί το κάθε σύστημα.

Τα δύο κύρια μοντέλα επικοινωνίας είναι το *σύγχρονο* (synchronous), το οποίο είναι γνωστό και ως μοντέλο *ραντεβού* (rendezvous), και το *ασύγχρονο* (asynchronous) ή *ανυπόμονο* (eager). Και στα δύο μοντέλα, αν ο παραλήπτης κάνει κλήση λήψης πριν ο αποστολέας αποστείλει το μήνυμα, σταματά τη λειτουργία του και περιμένει[2] μέχρι να έρθει το μήνυμα. Η διαφοροποίηση βρίσκεται στη συμπεριφορά του αποστολέα.

[2]Θα χρησιμοποιούμε τον όρο «μπλοκάρει», χάρη ευκολίας.



Σχήμα 5.1 *Σύγχρονη επικοινωνία: (α) ο αποστολέας έστειλε το μήνυμα πριν ο παραλήπτης κάνει κλήση λήψης και (β) ο παραλήπτης βρίσκεται ήδη σε κατάσταση λήψης όταν ο αποστολέας στέλνει το μήνυμα.*



Σχήμα 5.2 *Ασύγχρονη επικοινωνία: ο αποστολέας στέλνει άμεσα το μήνυμα χωρίς να τον ενδιαφέρει η κατάσταση του παραλήπτη, και συνεχίζει τη λειτουργία του.*

Στο σύγχρονο μοντέλο, αν ο αποστολέας αποστείλει το μήνυμα νωρίς, θα *μπλοκάρει* μέχρι ο παραλήπτης εκτελέσει κλήση λήψης. Μόλις συμβεί αυτό, τότε μπορεί να συνεχίσει τη λειτουργία του. Στην περίπτωση που ο παραλήπτης είχε ήδη προλάβει να κάνει κλήση λήψης, δεν θα υπάρχει επιπλέον καθυστέρηση στον αποστολέα και το μήνυμα τα μεταφερθεί άμεσα. Οι δύο περιπτώσεις φαίνονται στο Σχ. 5.1. Στην περίπτωση (α), η πηγή κάνει αίτημα αποστολής και σταματά τη λειτουργία της, περιμένοντας μέχρι να αποφασίσει ο προορισμός να κάνει λήψη. Αμέσως τότε, υπάρχει μία ενημέρωση από τον προορισμό προς την πηγή (acknowledgement) ότι μπορεί να προχωρήσει στην μετακίνηση των δεδομένων του μηνύματος, την οποία και κάνει. Οι δύο διεργασίες συνεχίζουν από εκεί με μετά κανονικά τη λειτουργία τους. Στην περίπτωση (β), αμέσως μόλις γίνει το αίτημα αποστολής, ο προορισμός ενημερώνει ότι είναι ήδη έτοιμος, και η πηγή αποστέλλει αμέσως το μήνυμα, χωρίς να χάσει άλλο χρόνο.

Στο ασύγχρονο μοντέλο, από την άλλη, η πηγή πάντα στέλνει το μήνυμά της *χωρίς να την ενδιαφέρει αν ο προορισμός είναι έτοιμος να το παραλάβει*, και συνεχίζει αμέσως

τη λειτουργία της χωρίς να μπλοκάρει ποτέ. Προκειμένου να γίνει αυτό, είναι απαραίτητη η ύπαρξη αποθηκευτικού χώρου (buffers), όπου θα φυλάσσεται το μήνυμα που έστειλε η πηγή με ασφάλεια μέχρι ο προορισμός να το παραλάβει. Η κατάσταση απεικονίζεται στο Σχ. 5.2. Αποτελεί σημαντικό θέμα πού θα βρίσκονται οι buffers αυτοί, αλλά δεν θα μας απασχολήσει εδώ. Αυτό που πρέπει να μας μείνει, είναι ότι η ασύγχρονη επικοινωνία αποτελεί την πιο συνηθισμένη μορφή επικοινωνίας στην πράξη. Είναι το εξ ορισμού μοντέλο που χρησιμοποιεί το MPI, και μάλιστα αναλαμβάνει αυτόματα, χωρίς να εμπλακεί ο προγραμματιστής, τη δημιουργία και διαχείριση των buffers. Είναι χαρακτηριστικό ότι την αποκαλεί *τυπική* επικοινωνία (standard mode), και είναι αυτή που χρησιμοποιείται όταν καλούμε την `MPI_Send()`.

Το ασύγχρονο μοντέλο έχει το εμφανές (και σημαντικό) πλεονέκτημα ότι ο αποστολέας δεν καθυστερεί καθόλου τη λειτουργία του, κάτι που δεν μπορεί να γίνει στο σύγχρονο μοντέλο. Επομένως, το ασύγχρονο μοντέλο δίνει δυνατότητα αυξημένων επιδόσεων, τουλάχιστον από την πλευρά του αποστολέα. Επιπρόσθετα, ας μην ξεχνάμε ότι το μοντέλο των ραντεβού περιλαμβάνει και μία φάση συνεννόησης (αίτημα και acknowledgement) που επιβαρύνουν ακόμα περισσότερο την επικοινωνιακή καθυστέρηση. Το τίμημα, όμως, για τις αυξημένες επιδόσεις είναι η ανάγκη ύπαρξης buffers. Αν προσέξει κανείς, στο σύγχρονο μοντέλο, από τη στιγμή που η πηγή γνωρίζει ότι ο προορισμός έχει εκτελέσει κλήση λήψης, μπορεί να προωθήσει τα δεδομένα του μηνύματος απευθείας στον χώρο που προσδιορίζει ο παραλήπτης. Έτσι, δεν υπάρχει ανάγκη για ενδιάμεσους buffers κάτι που συνεπάγεται οικονομία σε πόρους αλλά και ταχύτητα, καθώς δεν υπάρχει ο φόρτος της διαχείρισής τους. Ίσως, όμως, το πιο σημαντικό πλεονέκτημα των ραντεβού, είναι το γεγονός ότι οι επικοινωνίες είναι *προβλέψιμες*, καθώς τα δύο μέρη ολοκληρώνουν τις λειτουργίες τους πάντα μαζί, γνωρίζοντας καθένας σε ποια κατάσταση βρίσκεται ο άλλος. Αντίθετα, στο ασύγχρονο μοντέλο, ο αποστολέας δεν μπορεί να γνωρίζει σε ποια φάση βρίσκεται ο παραλήπτης και αντίστροφα. Αυτό, κάνει τη λειτουργία των προγραμμάτων κατά μεγάλο μέρος απρόβλεπτη, και σε πολύπλοκες εφαρμογές εισάγει δυσκολίες στην αποσφαλμάτωση και στο να εγγυηθεί κάποιος την ορθότητα της λειτουργίας. Παρ' όλα αυτά, οι επιδόσεις είναι αυτές που επιβάλλουν στις περισσότερες περιπτώσεις τη χρήση των ασύγχρονων επικοινωνιών και έτσι, οι ασύγχρονες επικοινωνίες συναντώνται συχνότερα στην πράξη.

Κλείνοντας αυτή την ενότητα, αξίζει να σημειώσουμε ότι το MPI, εκτός από το ασύγχρονο, προσφέρει στον προγραμματιστή και το σύγχρονο επικοινωνιακό μοντέλο των ραντεβού. Για να το χρησιμοποιήσει κάποιος, αντί της `MPI_Send()`, αρκεί να καλέσει τη συνάρτηση `MPI_Rsend()`, με τις ίδιες ακριβώς παραμέτρους. Πέρα από αυτό, παρέχονται παραλλαγές στα δύο επικοινωνιακά μοντέλα, οι οποίες ξεφεύγουν κατά πολύ από τα ενδιαφέροντά μας εδώ. Η λήψη σε οποιοδήποτε μοντέλο γίνεται πάντα με την `MPI_Recv()`. Τέλος, σε παρακάτω ενότητες, θα έχουμε την ευκαιρία να δούμε επικοινωνίες με εντελώς διαφορετική φιλοσοφία.

5.3 Παραδείγματα εφαρμογών και τεχνικών

Με βάση τις λίγες προγραμματιστικές δομές που είδαμε στην Ενότητα 5.1, είμαστε σε θέση να προγραμματίσουμε αμέσως τις πρώτες μας εφαρμογές. Οι αλγόριθμοι που θα χρησιμοποιήσουμε είναι παρόμοιοι με αυτούς που είδαμε σε αντίστοιχα παραδείγματα στο Κεφάλαιο 4.

Για κάποιον που έχει μελετήσει το προηγούμενο κεφάλαιο, τα πράγματα δεν θα πρέπει να φαίνονται ιδιαίτερα περίπλοκα. Αυτό είναι απόρροια του γεγονότος ότι και στο μοντέλο κοινού χώρου διευθύνσεων, ο προγραμματισμός γίνεται με χρήση πολλαπλών οντοτήτων εκτέλεσης, μόνο που αυτές είναι σχεδόν πάντα διεργασίες. Ό,τι τεχνικές χρησιμοποιήσαμε για το μοίρασμα της εργασίας στο μοντέλο του προηγούμενου κεφαλαίου, ίδιες χρησιμοποιούνται και στο μοντέλο μεταβίβασης μηνυμάτων. Επομένως, ένας σειριακός υπολογισμός με βρόχους μπορεί να παραλληλοποιηθεί ως μία συλλογή διεργασιών χρησιμοποιώντας τεχνικές, όπως:

- τμηματική δρομολόγηση,
- διάσπαση βρόχου,
- διαχωρισμό σκακιέρας,
- αυτοδρομολόγηση.

Η μόνη διαφορά, η οποία είναι καθοριστική, είναι ότι οι διεργασίες δεν μπορούν να έχουν κοινόχρηστες μεταβλητές. Επομένως, οι διεργασίες δεν μπορούν να έχουν άμεση πρόσβαση στα δεδομένα, αφού αυτά δεν αποθηκεύονται σε κάποιο κοινό χώρο. Θα υπάρχει σχεδόν πάντα μία συγκεκριμένη διεργασία η οποία θα αναλαμβάνει την αρχικοποίηση των δεδομένων και, ίσως, την επικοινωνία με τον χρήστη. Αυτή θα «μοιράζει» τους υπολογισμούς και τα δεδομένα στις υπόλοιπες και στο τέλος, θα «συλλέγει» από αυτές τα αποτελέσματα.

5.3.1 Υπολογισμός της σταθεράς $\pi = 3.14159...$

Ως πρώτο παράδειγμα προγραμματισμού στο μοντέλο μας, θα δούμε πώς μπορούμε να υπολογίσουμε το π χρησιμοποιώντας την αριθμητική ολοκλήρωση, όπως την εφαρμόσαμε και στο προηγούμενο κεφάλαιο.

Προκειμένου να παραλληλοποιήσουμε τον σειριακό κώδικα, θα μοιράσουμε τους υπολογισμούς σε διεργασίες. Για τον σκοπό αυτό, θα χρησιμοποιήσουμε την τεχνική της διάσπασης βρόχου. Έστω, λοιπόν ότι `N` είναι το πλήθος των όρων του αθροίσματος και `W` το πλάτος του κάθε διαστήματος ($W = 1/N$). Η υλοποίηση θα πρέπει να γίνει περίπου όπως παρακάτω:

```
double result = 0.0;
int    i;

for (i = myid; i < N; i += nproc)
  result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
```

Κάθε διεργασία βρίσκει την ταυτότητά της (myid) μέσω κλήσης στη συνάρτηση MPI_Comm_rank(), και υπολογίζει τους όρους που της αντιστοιχούν, αθροίζοντάς τους στην μεταβλητή result. Το βήμα αύξησης στο βρόχο, όπως γνωρίζουμε, θα πρέπει να είναι ίσο με τον αριθμό των διεργασιών (nproc), κάτι που το βρίσκουμε εύκολα με κλήση στην MPI_Comm_size().

Μέχρι αυτό το σημείο, δεν συναντήσαμε ουσιαστικά τίποτε καινούριο. Τι γίνεται, όμως, με τον υπολογισμό του π συνολικά; Είμαστε στην κατάσταση όπου κάθε διεργασία έχει υπολογίσει τη δική της συνεισφορά στο π (result). Προκειμένου να ολοκληρωθεί ο υπολογισμός, θα πρέπει κάποια διεργασία να «συλλέξει» όλα τα μερικά αυτά αθροίσματα και να τα προσθέσει. Επομένως, επιλέγουμε μία διεργασία, έστω τη διεργασία 0, η οποία πρέπει να λάβει μηνύματα από όλες τις άλλες· τα μηνύματα θα περιέχουν την τιμή της μεταβλητής result από κάθε διεργασία και η διεργασία 0 θα τα προσθέσει όλα προκειμένου να βρει την τιμή του π.

Ο ολοκληρωμένος κώδικας δίνεται στο Σχ. 5.3, το οποίο μας δίνει την ευκαιρία να δούμε σε ολοκληρωμένη μορφή ένα τυπικό πρόγραμμα γραμμένο σε MPI. Αρχικά, είναι απαραίτητο να χρησιμοποιηθεί το αρχείο επικεφαλίδων mpi.h (γραμμή 1). Επίσης, κάθε πρόγραμμα πρέπει να ξεκινά με κλήση στη συνάρτηση MPI_Init() (γραμμή 8) και να τερματίζει με κλήση στην MPI_Finalize() (γραμμή 38). Όταν ο χρήστης εκτελέσει το πρόγραμμα, θα καθορίσει και το συνολικό πλήθος διεργασιών. Θα δημιουργηθούν τότε πολλαπλές ανεξάρτητες διεργασίες, και όλες θα εκτελούν τον κώδικα στο Σχ. 5.3. Η γραμμή 9 δίνει σε κάθε διεργασία την ταυτότητά της.

Κάναμε το πρόγραμμα λίγο πιο πολύπλοκο, ζητώντας από τον χρήστη να δώσει το πλήθος των διαστημάτων που επιθυμεί. Η διεργασία 0 επιλέχθηκε να επικοινωνήσει με το χρήστη, όπως φαίνεται στις γραμμές 13–18. Προσέξτε ότι, οι υπόλοιπες διεργασίες δεν θα γνωρίζουν την τιμή που έδωσε ο χρήστης και, επομένως, είναι απαραίτητο να τους μεταβιβαστεί η τιμή του N (γραμμές 16–17). Έτσι, κάθε μία από τις υπόλοιπες θα πρέπει να περιμένει να το λάβει από τη διεργασία 0, και αυτό γίνεται στη γραμμή 20. Δώσαμε, τυπικά, ετικέτα με τιμή 0 στα μηνύματα αυτά.

Οι γραμμές 23–25 εκτελούνται από όλες τις διεργασίες. Φυσικά, η κάθε μία έχει διαφορετική ταυτότητα και, άρα, εκτελεί διαφορετικές επαναλήψεις, με βάση την διάσπαση βρόχου. Τέλος, κάθε διεργασία αποστέλλει το αποτέλεσμά της στη διεργασία 0 (γραμμή 36), η οποία περιμένει να λάβει όλα αυτά τα μερικά αποτελέσματα μέσα από ένα βρόχο (γραμμές 29–32). Τα αποτελέσματα που λαμβάνει (μεταβλητή temp) τα αθροίζει στη δική της μεταβλητή result και εκτυπώνει το τελικό αποτέλεσμα.

```
#include <mpi.h>

int main(int argc, char *argv[]) {
  double     W, result = 0.0, temp;
  int        N, i, myid, nproc;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Initialization */
  if (myid == 0) {
    printf("Enter number of divisions: ");
    scanf("%d", &N);
    for (i = 1; i < nproc; i++)
      MPI_Send(&N, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(&N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

  /* The actual computation */
  W = 1.0 / N;
  for (i = myid; i < N; i += nproc)
    result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);

  /* Gather results */
  if (myid == 0) {
    for (i = 1; i < nproc; i++) {
      MPI_Recv(&temp, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
      result += temp;
    }
    printf("pi = %lf", result);
  }
  else
    MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

Πρόγραμμα 5.3 *Απλός υπολογισμός της σταθεράς π με* MPI. *(mp-pi1.c)*

Βελτιστοποίηση

Προσέξτε, στον παραπάνω κώδικα, τον τρόπο με τον οποίο η διεργασία 0 λαμβάνει τα μηνύματα· πρώτα περιμένει να έρθει ένα μήνυμα από τη διεργασία 1, μετά ένα από τη

διεργασία 2, κλπ. Τι γίνεται, όμως, αν π.χ. τα μηνύματα δεν έρθουν με αυτή τη σειρά διότι πιθανώς κάποιοι κόμβοι του συστήματος καθυστέρησαν στην εκτέλεση των διεργασιών τους; Από προγραμματιστικής πλευράς δεν υπάρχει πρόβλημα—κανένα μήνυμα δεν χάνεται, απλά θα παραδοθεί όταν το ζητήσει η διεργασία 0, ακόμα και αν έχει φτάσει νωρίτερα από άλλα. Από πλευράς επιδόσεων και πόρων, όμως, ο παραπάνω κώδικας μπορεί να μην είναι πάντα ο καλύτερος δυνατός. Όπως γνωρίζουμε, οι εξ ορισμού επικοινωνίες στο MPI είναι ασύγχρονες, και τα μηνύματα φυλάσσονται σε buffers μέχρι να παραδοθούν. Αυτό σίγουρα σπαταλά και χρόνο για τη διαχείριση, αλλά και αποθηκευτικό χώρο, ιδιαίτερα αν υπάρξουν πολλά τέτοια μηνύματα.

Η καλύτερη στρατηγική είναι να παραλαμβάνονται τα μηνύματα με τη σειρά που καταφθάνουν στην διεργασία 0, έτσι ώστε να ελαχιστοποιηθούν τα προβλήματα που αναφέραμε. Η μόνη αλλαγή που απαιτείται για να γίνει αυτό, είναι η αντικατάσταση της κλήσης λήψης με την παρακάτω:

```
30        MPI_Recv(&temp, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
                   &status);
```

Πράγματι, όπως είπαμε στην Ενότητα 5.1, το αποτέλεσμα μίας τέτοιας κλήσης είναι η παραλαβή *οποιουδήποτε* μηνύματος έχει φθάσει (αρκεί να έχει ετικέτα 0), το οποίο είναι ακριβώς αυτό που θέλουμε. Ως γενική αρχή, μπορούμε να πούμε ότι, εκτός και αν η σειρά των μηνυμάτων παίζει κάποιο σημαντικό ρόλο (π.χ. λόγω του σχεδιασμού του αλγορίθμου), η παραλαβή των μηνυμάτων είναι συνήθως καλύτερα να γίνεται με τη σειρά που αυτά καταφθάνουν.

5.3.2 Πίνακας επί διάνυσμα

Συνεχίζοντας τα παραδείγματά μας, θα δούμε τώρα πώς μπορούμε να υπολογίσουμε το γινόμενο ενός πίνακα επί ένα διάνυσμα χρησιμοποιώντας μεταβίβαση μηνυμάτων. Όπως και στο Κεφάλαιο 4, έτσι και τώρα, ο σειριακός κώδικας για το γινόμενο του πίνακα `A[N][N]` επί το διάνυσμα `v[N]`:

```
for (i = 0; i < N; i++) {
  for (sum = 0.0, j = 0; j < N; j++)
    sum += A[i][j]*v[j];
  res[i] = sum;
}
```

μπορεί να οδηγήσει σε διεργασίες οι οποίες προκύπτουν είτε από τον παραλληλισμό του βρόχου `i` (με τμηματική δρομολόγηση ή διάσπαση βρόχου) είτε από ταυτόχρονο παραλληλισμό και των δύο βρόχων (`i` και `j`) μαζί.

Για λόγους απλότητας, θα υποθέσουμε ότι διαθέτουμε λιγότερους από `N` κόμβους στο σύστημά μας, οπότε η τμηματική δρομολόγηση στο βρόχο του `i` είναι μία αποδοτική

προσέγγιση. Υποθέτοντας ότι το πλήθος των διεργασιών είναι nproc, και ότι διαιρεί ακριβώς το N, ο κώδικας που αντιστοιχεί σε κάθε διεργασία θα είναι, επομένως, ο εξής:

```
WORK = N / nproc;  /* Rows per process (assuming nproc divides N) */
sum = 0.0;

for (i = 0; i < WORK; i++) {
  for (j = 0; j < N; j++)
    sum += A[myid*WORK+i][j]*v[j];
  /* sum will contain the (myid*WORK+i)-th element of result */
```

Κάθε διεργασία αναλαμβάνει WORK στοιχεία του αποτελέσματος και, συγκεκριμένα, τα στοιχεία myid*WORK, myid*WORK+1, …, (myid+1)*WORK-1, όπου myid είναι η ταυτότητα της διεργασίας. Κάθε στοιχείο που υπολογίζει, θα πρέπει, όπως έγινε και στην Ενότητα 5.3.1, να το στέλνει σε μία διεργασία (έστω τη διεργασία 0) η οποία θα αναλάβει την συλλογή των στοιχείων του τελικού αποτελέσματος, ώστε πιθανώς να τα παρουσιάσει στο χρήστη. Επομένως, ο κώδικας θα πρέπει να συνεχίζει με την αποστολή των μηνυμάτων και την παραλαβή τους από τη διεργασία 0.

Όμως, η κάθε διεργασία θα στείλει πολλά (WORK στο πλήθος) μηνύματα στη διεργασία 0. Θα πρέπει με κάποιο τρόπο, σε κάθε μήνυμα που μεταβιβάζεται, να προσδιορίζεται και ποιο από τα στοιχεία του αποτελέσματος περιέχεται στο μήνυμα. Εδώ, φαίνεται καθαρά η χρησιμότητα της ετικέτας των μηνυμάτων. Η πηγή του μηνύματος δίνει μία ετικέτα, η οποία ως τιμή έχει τον αύξοντα αριθμό του στοιχείου που περιλαμβάνεται στο μήνυμα. Επομένως, ο βρόχος του i ολοκληρώνεται ως εξής:

```
WORK = N / nproc;  /* Rows per process (assuming nproc divides N) */
sum = 0.0;

for (i = 0; i < WORK; i++) {
  for (j = 0; j < N; j++)
    sum += A[myid*WORK+i][j]*v[j];
  /* sum will contain the (myid*WORK+i)-th element of result */
  MPI_Send(&sum, 1, MPI_DOUBLE, 0, myid*WORK+i, MPI_COMM_WORLD);
}
```

Μία είναι μόνο η λεπτομέρεια που μένει προκειμένου να ολοκληρωθεί το πρόγραμμά μας. Πώς γνωρίζει η κάθε διεργασία τα στοιχεία του A και του v; Η απάντηση είναι ότι αρχικά δεν μπορεί να τα γνωρίζει, αλλά θα πρέπει να τα μάθει *παραλαμβάνοντάς τα από τη διεργασία 0*. Η διεργασία 0 θα επωμιστεί και αυτό το ρόλο: θα αρχικοποιήσει τον πίνακα A και το διάνυσμα v (π.χ. είτε επικοινωνώντας με τον χρήστη είτε διαβάζοντας τα στοιχεία από κάποιο αρχείο) και στη συνέχεια, θα μοιράσει τα κατάλληλα στοιχεία σε κάθε διεργασία. Συγκεκριμένα, στη διεργασία i θα πρέπει να στείλει τις WORK γραμμές του A που της αντιστοιχούν, αλλά και το διάνυσμα v.

Το τελικό πρόγραμμα δίνεται στο Σχ. 5.4. Για χάρη συντομίας, δίνουμε μόνο τη συνάρτηση που κάνει τους υπολογισμούς. Το κυρίως πρόγραμμα έχει κάνει την αρχικοποίηση

```
#define N 1000          /* Matrix dimension */

void matrix_times_vector(int myid, int nproc) {
  int        i, j;
  int        WORK = N / nproc;      /* Rows per process */
  double     sum = 0.0, v[N];
  MPI_Status status;

  if (myid == 0) {                  /* Process 0 */
    double A[N][N], res[N];

    initialize_elements(A, v);      /* Some kind of initalization */
    for (i = 1; i < nproc; i++) {   /* Distribute matrix & vector */
      MPI_Send(A[i*WORK], WORK*N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      MPI_Send(v, N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
    }
    for (i = 0; i < WORK; i++) {    /* Doing my own work */
      for (sum = 0.0, j = 0; j < N; j++)
        sum += A[i][j]*v[j];
      res[i] = sum;
    }
    /* Gather all result elements from other processes */
    for (i = WORK; i < WORK*nproc; i++) {
      MPI_Recv(&sum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
               MPI_COMM_WORLD, &status);
      res[ status.MPI_TAG ] = sum;  /* Tag has element position! */
    }
    show_result(res);               /* Display the end result */
  }
  else {                            /* All other processes */
    double B[WORK][N];              /* See remarks; C99-only */

    MPI_Recv(B, WORK*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(v, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
    for (i = 0; i < WORK; i++) {
      for (sum = 0.0, j = 0; j < N; j++)
        sum += B[i][j]*v[j];
      /* sum will contain the (myid*WORK+i)-th element of result */
      MPI_Send(&sum, 1, MPI_DOUBLE, 0, myid*WORK+i, MPI_COMM_WORLD);
    }
  }
}
```

Πρόγραμμα 5.4 *Αρχική έκδοση υπολογισμού πίνακα επί διάνυσμα με χρήση* MPI. *(mp-mxv1.c)*

του MPI και έχει βρει την ταυτότητα της διεργασίας και το συνολικό πλήθος διεργασιών, τα οποία και δίνονται ως παράμετροι στην `matrix_times_vector()`. Οι γραμμές 9–29 αφορούν τη διεργασία 0, η οποία αρχικοποιεί τον πίνακα `A` και το διάνυσμα `v`, και διανέμει τις κατάλληλες γραμμές του πίνακα, καθώς και όλο το διάνυσμα στις υπόλοιπες διεργασίες. Στη συνέχεια, υπολογίζει το δικό της τμήμα του αποτελέσματος, δηλαδή τα πρώτα `WORK` στοιχεία του `res`, και τέλος παραλαμβάνει τα υπόλοιπα στοιχεία από τις άλλες διεργασίες. Οι διεργασίες αυτές εκτελούν τις γραμμές 30–41, όπου αρχικά παραλαμβάνουν τις γραμμές που τους αντιστοιχούν και τις τοποθετούν στον πίνακα `B`, υπολογίζουν τα ανάλογα στοιχεία του αποτελέσματος και κάθε ένα από αυτά το στέλνουν στην διεργασία 0, με βάση την τεχνική της ετικέτας, που είπαμε παραπάνω.

Πριν προχωρήσουμε παρακάτω, είναι απαραίτητο να γίνουν μερικές παρατηρήσεις στο πρόγραμμα αυτό:

1. Στη γραμμή 14, η διεργασία 0 κατασκευάζει, για κάθε παραλήπτη `i`, ένα μήνυμα που αποτελείται από `WORK` συνεχόμενες γραμμές του `A` (αρχίζοντας από τη γραμμή `A[i*WORK]`), μεγέθους `WORK*N` στοιχείων, τύπου `double`. Εδώ, χρησιμοποιήσαμε το γεγονός ότι η C αποθηκεύει τον πίνακα στη μνήμη κατά γραμμές. Εάν προγραμματίζαμε σε Fortran (όπου η αποθήκευση γίνεται κατά στήλες), θα έπρεπε να προετοιμάζαμε το μήνυμα σε κάποιο άλλο αποθηκευτικό χώρο, σχηματίζοντας τις κατάλληλες γραμμές.

2. Η κάθε διεργασία δεν χρειάζεται χώρο για την αποθήκευση ολόκληρου του πίνακα `A`, παρά μόνο για τις `WORK` γραμμές του που θα λάβει. Αυτές αποθηκεύονται στον πίνακα `B` ο οποίος θα έχει `WORK` γραμμές και `N` στήλες. Με αυτόν τον τρόπο, απλοποιούνται και οι υπολογισμοί, αφού οι γραμμές `myid*WORK`, `myid*WORK+1`, ..., `(myid+1)*WORK-1` του `A` που απαιτεί η διεργασία, είναι οι γραμμές 0, 1, ..., `WORK-1` του `B`. Εδώ, πρέπει να σημειωθεί ότι η δήλωση του πίνακα `B` δεν είναι συμβατή με την κλασική έκδοση της ANSI C, καθώς το μέγεθός του δίνεται από μία μεταβλητή (`WORK`)—η έκδοση C99 το επιτρέπει. Παρ' όλα αυτά, το αφήνουμε έτσι, χάρη απλότητας. Στους πηγαίους κώδικες που συνοδεύουν το βιβλίο, η δήλωση είναι συμβατή με την κλασική C και γίνεται δυναμική εκχώρηση μνήμης για τον πίνακα `B`.

3. Υποθέσαμε ότι υπάρχουν δύο συναρτήσεις (συγκεκριμένα οι `initialize_elements()` και `show_result()`) οι οποίες αρχικοποιούν τα στοιχεία των `A` και `v` και δίνουν στον χρήστη το τελικό αποτέλεσμα, αντίστοιχα. Τις συναρτήσεις αυτές τις καλεί, φυσικά, η διεργασία 0.

Βελτιστοποίηση

Στο μοντέλο μεταβίβασης μηνυμάτων, προσπαθούμε (όσο γίνεται) να αποφεύγουμε τις πολλές και συχνές επικοινωνίες, λόγω του ότι αποτελούν συνήθως τη σημαντικότερη πηγή καθυστερήσεων. Αν πρόκειται να στείλουμε κάποια δεδομένα, το ένα μετά το άλλο, είναι σε γενικές γραμμές προτιμητέο να τα «πακετάρουμε» όλα σε ένα μεγάλο μήνυμα και να τα στείλουμε μαζί, παρά να τα στείλουμε ξεχωριστά, με πολλά μικρότερα μηνύματα. Σε τυπικά συστήματα, όπως συνήθεις υπολογιστικές συστάδες, η τεχνική μειώνει σημαντικά το χρόνο που σπαταλάται σε επικοινωνίες και μπορεί να αυξήσει ανάλογα και τις επιδόσεις. Σε συστήματα που στηρίζονται σε δίκτυα πολύ χαμηλής καθυστέρησης (low latency), τα κέρδη από την τεχνική αυτή μπορεί να μην είναι μεγάλα, όμως ακόμα και εκεί, μπορεί να εξοικονομήσουν χρόνο.

Στο πλαίσιο αυτό, μία βελτιστοποίηση που θα μπορούσαμε να κάνουμε στον κώδικα του Σχ. 5.4, είναι να συγκεντρώναμε όλα τα στοιχεία που υπολογίζει η κάθε διεργασία σε ένα μήνυμα, αντί να στείλει `WORK` διαφορετικά μηνύματα στη διεργασία 0. Προκειμένου να γίνει αυτό, στις γραμμές 35–39, θα πρέπει το κάθε στοιχείο που υπολογίζεται να αποθηκεύεται τοπικά σε ένα διάνυσμα, και στο τέλος το διάνυσμα αυτό να αποστέλλεται, με όλα τα `WORK` στοιχεία του, σε ένα μήνυμα. Η ετικέτα του μηνύματος θα αναφέρει τη θέση του πρώτου από τα στοιχεία (τα υπόλοιπα είναι σε συνεχόμενες θέσεις).

Το βελτιστοποιημένο πρόγραμμα δίνεται στο Σχ. 5.5. Από τη στιγμή που κάθε διεργασία στέλνει μόνο ένα μήνυμα, και όχι `WORK` μηνύματα, η διεργασία 0 θα λάβει συνολικά `nproc-1` μηνύματα (γραμμή 23). Κάθε μήνυμα που λαμβάνει στο `mypart` έχει `WORK` στοιχεία και τα τοποθετεί στη σωστή θέση του `res`, χρησιμοποιώντας την ετικέτα του μηνύματος (γραμμές 26–27).

5.3.3 Απολογισμός του βασικού μοντέλου

Στο σημείο αυτό, είναι σκόπιμο να κάνουμε έναν μικρό απολογισμό του μοντέλου μεταβίβασης μηνυμάτων, στη βασική του μορφή, όπως το είδαμε μέχρι τώρα. Θα πρέπει να έχει γίνει φανερό ότι ο προγραμματισμός στο μοντέλο που μελετάμε σε αυτό το κεφάλαιο, οδηγεί σε αρκετά πιο μακροσκελή προγράμματα (συγκρίνετε με τα αντίστοιχα προγράμματα στο μοντέλο κοινόχρηστου χώρου διευθύνσεων). Σκεφτείτε, επίσης, ότι το πρόγραμμα για τον υπολογισμό του γινομένου πίνακα επί διάνυσμα που δώσαμε, υποθέτει ότι το πλήθος των διεργασιών διαιρεί *ακριβώς* τη διάσταση του πίνακα. Αν αυτό δεν ισχύει, που μόνο απίθανο δεν είναι, το πρόγραμμα στο Σχ. 5.5 θα ήταν ακόμα πιο ογκώδες και περίπλοκο. Η διαμοίραση και διαχείριση των δεδομένων απαιτεί μεγάλη προσοχή και ο κώδικας, τελικά, γίνεται γρήγορα δυσανάγνωστος και λιγότερο διαχειρίσιμος.

Αυτό είναι το βασικό επιχείρημα εναντίον του μοντέλου μεταβίβασης μηνυμάτων, και γι' αυτό πολλοί το αποφεύγουν. Όμως, σε μερικές περιπτώσεις, το μοντέλο αυτό μπο-

```
1  #define N 1000         /* Matrix dimension */
2
3  void matrix_times_vector(int myid, int nproc) {
4    int        i, j;
5    int        WORK = N / nproc;      /* Rows per process */
6    double     sum = 0.0, v[N], mypart[N];
7    MPI_Status status;
8
9    if (myid == 0) {                  /* Process 0 */
10     double A[N][N], res[N];
11
12     initialize_elements(A, v);       /* Some kind of initalization */
13     for (i = 1; i < nproc; i++) {    /* Distribute matrix & vector */
14       MPI_Send(A[i*WORK], WORK*N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
15       MPI_Send(v, N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
16     }
17     for (i = 0; i < WORK; i++) {     /* Doing my own work */
18       for (sum = 0.0, j = 0; j < N; j++)
19         sum += A[i][j]*v[j];
20       res[i] = sum;
21     }
22     /* Gather results from other processes */
23     for (i = 1; i < nproc; i++) {
24       MPI_Recv(mypart, WORK, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
25                MPI_COMM_WORLD, &status);
26       for (j = 0; j < WORK; j++)     /* Place elements correctly */
27         res[ j + status.MPI_TAG ] = mypart[j];
28     }
29     show_result(res);                /* Display the end result */
30   }
31   else {                             /* All other processes */
32     double B[WORK][N];               /* See remarks; C99-only */
33
34     MPI_Recv(B, WORK*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
35     MPI_Recv(v, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
36     for (i = 0; i < WORK; i++) {
37       for (sum = 0.0, j = 0; j < N; j++)
38         sum += B[i][j]*v[j];
39       mypart[i] = sum;               /* Keep element */
40     }
41     /* myid*WORK is my the first of the computed elements */
42     MPI_Send(mypart, WORK, MPI_DOUBLE, 0, myid*WORK, MPI_COMM_WORLD);
43   }
44 }
```

Πρόγραμμα 5.5 *Δεύτερη εκδοχή του υπολογισμού πίνακα επί διάνυσμα σε* MPI, *με βελτιστοποίηση στις επικοινωνίες. (mp-mxv2.c)*

ρεί, με κατάλληλη (και επίπονη) βελτιστοποίηση των επικοινωνιών, να δώσει προγράμματα αποδοτικότερα απ' ό,τι θα ήταν δυνατόν με το μοντέλο κοινόχρηστου χώρου διευθύνσεων. Το γεγονός ότι κάθε επικοινωνία την καθορίζει και την ελέγχει ο προγραμματιστής, αν και αρνητικό για πολλούς, για άλλους είναι ο μόνος τρόπος να αντληθούν οι όποιες επιταχύνσεις μπορεί να αποδώσει το διαθέσιμο σύστημα. Επίσης, ο άμεσος καθορισμός των επικοινωνιών προσφέρει, με σχετική ακρίβεια, μία εκτίμηση για τη χρονική συμπεριφορά των προγραμμάτων. Και αυτό είναι σε αντίθεση με το μοντέλο κοινής μνήμης, όπου ο συναγωνισμός των διεργασιών για τις κοινές μεταβλητές έχει απροσδιόριστη συμπεριφορά και μη προβλέψιμες καθυστερήσεις. Όπως και να έχει το θέμα, το μοντέλο αυτό είναι μονόδρομος—τουλάχιστον μέχρι σήμερα—για προγραμματισμό συστάδων. Επιπρόσθετα, επειδή εν γένει τα μεγάλα υπολογιστικά συστήματα στηρίζονται σε κατανεμημένη μνήμη, όσο δύσκολη ή «χαμηλού» επιπέδου και να είναι, η μεταβίβαση μηνυμάτων με χρήση του MPI είναι το πιο ευρέως χρησιμοποιούμενο μοντέλο παράλληλου προγραμματισμού υψηλών επιδόσεων.

5.4 Συλλογικές επικοινωνίες

Στην ενότητα αυτή θα δούμε πόσο χρήσιμες μπορούν να φανούν κάποιες επιπλέον ευκολίες που μπορεί να παρέχονται στο μοντέλο μεταβίβασης μηνυμάτων, παρότι δεν είναι υποχρεωτικό να προσφέρονται. Συγκεκριμένα, θα δούμε τις θετικές επιπτώσεις που έχουν οι κλήσεις που υλοποιούν τις λεγόμενες *συλλογικές επικοινωνίες* (collective communications) μεταξύ διεργασιών. Το MPI διαθέτει πολύ μεγάλη ποικιλία από τέτοιες κλήσεις και είναι, σε σημαντικό βαθμό, υπεύθυνο για την καθιέρωσή τους ως άκρως επιθυμητό συστατικό του μοντέλου που μελετάμε στο κεφάλαιο αυτό.

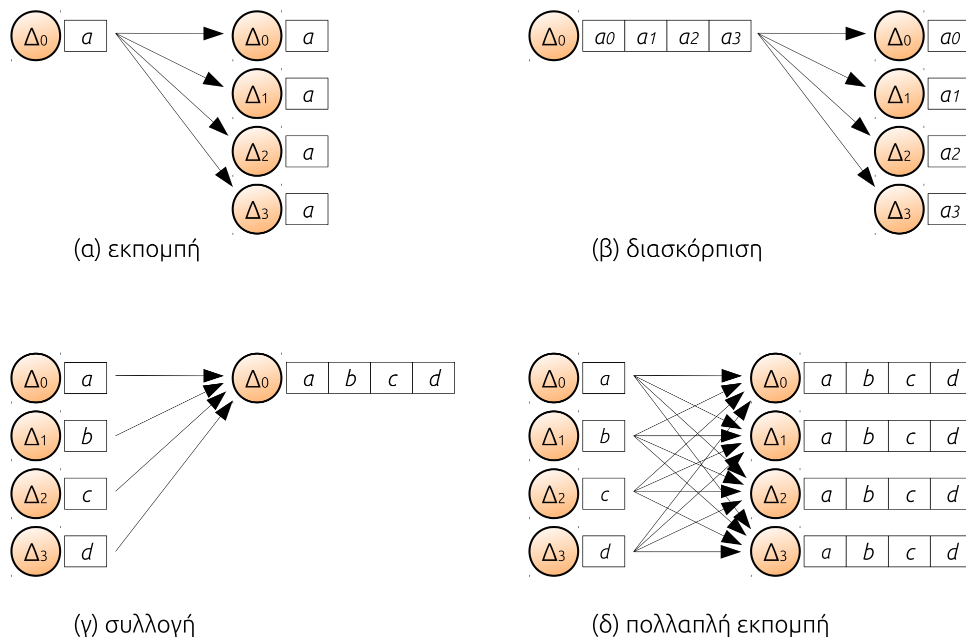
Είναι πολλές οι περιπτώσεις που σε κάποιες από τις επικοινωνίες εμπλέκονται παραπάνω από δύο διεργασίες, δηλαδή δεν υπάρχει μόνο ένας αποστολέας και ένας παραλήπτης. Ένα απλό παράδειγμα μπορούμε να δούμε στο γινόμενο πίνακα επί διάνυσμα, που υλοποιήσαμε στην Ενότητα 5.3.2. Στη συγκεκριμένη εφαρμογή, εκτός των άλλων, η διεργασία 0 θα πρέπει να στείλει το διάνυσμα `v[]` σε όλες τις υπόλοιπες διεργασίες, έστω n στο πλήθος. Η απλοϊκή μέθοδος που χρησιμοποιήσαμε στέλνει n διαφορετικά μηνύματα, ένα σε κάθε διεργασία. Αν σκεφτούμε, όμως, λίγο περισσότερο, θα δούμε ότι κάτι τέτοιο δεν είναι το καλύτερο που θα μπορούσαμε να κάνουμε. Αν, για παράδειγμα, όλοι οι επεξεργαστές ήταν συνδεδεμένοι σε έναν δίαυλο ή προγραμματίζαμε σε συστάδα ή τοπικό δίκτυο με διασύνδεση τύπου ethernet (η οποία είναι πρακτικά διασύνδεση διαύλου), μόλις *ένα* μήνυμα θα ήταν αρκετό για να λάβουν το `v[]` όλοι οι επεξεργαστές / διεργασίες, αφού όλοι «παρακολουθούν» το κοινόχρηστο μέσο διασύνδεσης. Ακόμα και αν η διασύνδεση ήταν διαφορετική, π.χ. είχαμε τοπολογία υπερκύβου, κάθε μήνυμα θα έπρεπε να διασχίσει κάποια διαδρομή

μέσα στο δίκτυο προκειμένου να φτάσει από τον κόμβο 0 (που θεωρούμε ότι εκτελεί τη διεργασία 0) σε κάποιον άλλο κόμβο. Όμως, κατά τη διάρκεια αυτής της διαδρομής, όλοι οι ενδιάμεσοι κόμβοι θα μπορούσαν να παραλάβουν αυτό το μήνυμα, χωρίς να χρειαστεί να αποσταλούν ξεχωριστά μηνύματα για καθέναν από αυτούς. Αν μάλιστα, χρησιμοποιούσε κανείς ειδικά δέντρα, επάνω στα οποία δρομολογούσε τις αποστολές, μπορεί να αποδειχτεί ότι θα χρειάζονταν μόνο $\log n$ βήματα, ώστε και οι n διεργασίες να ενημερώνονταν για το μήνυμα.

Επομένως, σε ορισμένες περιπτώσεις επικοινωνιών όπου εμπλέκονται όλες οι διεργασίες, η αποστολή προσωπικών / ιδιωτικών μηνυμάτων δεν είναι πάντα η καλύτερη λύση. Αντίθετα, το σύστημα ίσως, μπορεί να διανείμει τα μηνύματα με αποδοτικότερο τρόπο, αρκεί να γνωρίζει περί ποιου είδους επικοινωνίας πρόκειται. Οι επικοινωνίες αυτές ονομάζονται συλλογικές επειδή αφορούν όλες τις εμπλεκόμενες διεργασίες και κάνουν εντελώς φυσιολογικά την εμφάνισή τους στη μεγάλη πλειονότητα των εφαρμογών. Πιθανοί τύποι τέτοιων επικοινωνιών απεικονίζονται στο Σχ. 5.6 (όπου έχουμε υποθέσει ότι υπάρχουν 4 διεργασίες συνολικά) και είναι οι παρακάτω:

- *εκπομπή* (broadcasting) όπου μία διεργασία θέλει να στείλει το *ίδιο* μήνυμα σε όλες τις άλλες,
- *διασκόρπιση* (scattering) όπου μία διεργασία πρέπει να στείλει ένα *διαφορετικό* μήνυμα σε κάθε μία από τις υπόλοιπες,
- *συλλογή* (gathering) όπου μία διεργασία θα λάβει ένα ξεχωριστό μήνυμα από κάθε άλλη διεργασία,
- *πολλαπλή εκπομπή* (multinode ή all-to-all broadcasting) όπου κάθε διεργασία κάνει εκπομπή. Το MPI την ονομάζει «allgather».

Πρέπει να σημειώσουμε ότι αυτές δεν είναι οι μόνες περιπτώσεις. Υπάρχουν και πολλά άλλα σενάρια, που εμφανίζονται λιγότερο ή περισσότερο συχνά. Ένα από τα χαρακτηριστικότερα είναι οι λειτουργίες υποβίβασης (reductions), που είδαμε στο Κεφάλαιο 4 ότι παρέχονται και από το OpenMP. Μια λειτουργία υποβίβασης συνδυάζει στοιχεία από όλες τις διεργασίες για να υπολογίσει ένα τελικό αποτέλεσμα. Για παράδειγμα, η υποβίβαση αθροίσματος προσθέτει ένα δεδομένο από κάθε διεργασία και υπολογίζει το συνολικό άθροισμα. Η λειτουργία αυτή εμπεριέχει συλλογική επικοινωνία. Μπορεί να υλοποιηθεί ως μία επικοινωνία συλλογής από τη διεργασία 0, η οποία στη συνέχεια θα αθροίσει (σειριακά) τα στοιχεία που θα λάβει. Θα μπορούσε, όμως, επιμέρους πράξεις να γίνονται σε ενδιάμεσους κόμβους, καθώς τα δεδομένα μεταφέρονται προς τη διεργασία 0 και επομένως, η τελευταία να λάβει τελικά το συνολικό αποτέλεσμα, ολοκληρώνοντας τη διαδικασία πιθανώς ταχύτερα.



Σχήμα 5.6 *Μερικοί τύποι συλλογικών επικοινωνιών μεταξύ 4 διεργασιών,* $\Delta_0 - \Delta_3$.

5.4.1 Συλλογικές επικοινωνίες στο MPI

Το MPI προσφέρει ένα μεγάλο πλήθος συναρτήσεων για συλλογικές επικοινωνίες. Το ιδιαίτερο χαρακτηριστικό των συναρτήσεων αυτών είναι ότι πρέπει να *κληθούν* συλλογικά, δηλαδή, όλες οι συμμετέχουσες διεργασίες πρέπει να καλέσουν ακριβώς την ίδια συνάρτηση. Με άλλα λόγια, δεν υπάρχουν κάποιες διεργασίες που κάνουν ρητά αποστολή ή κάποιες που κάνουν ρητά λήψη μηνυμάτων.

Εκπομπή — Ξεκινώντας από την πιο απλή συλλογική επικοινωνία, αυτή της εκπομπής, όλες οι διεργασίες θα πρέπει να κάνουν την παρακάτω κλήση:

```
MPI_Bcast(buf, n, dtype, rootrank, MPI_COMM_WORLD);
```

Όπως και στις συνήθεις κλήσεις λήψης και αποστολής, `n` είναι το πλήθος των στοιχείων και `dtype` είναι ο τύπος των στοιχείων για το μήνυμα που θα εκπεμφθεί. Στην κλήση αυτή, όμως:

- `buf` είναι ο χώρος για μήνυμα: η διεργασία που εκπέμπει, εκεί έχει το μήνυμα που πρέπει να σταλεί. Για τις υπόλοιπες διεργασίες το `buf` είναι ο χώρος στον οποίο πρέπει να αποθηκευτεί το μήνυμα όταν παραληφθεί.

- `rootrank` είναι η ταυτότητα της διεργασίας που εκπέμπει το μήνυμα και πρέπει, προφανώς, όλες οι διεργασίες να περάσουν εδώ την ίδια τιμή.

Όλες οι διεργασίες πρέπει να εκτελέσουν την κλήση στην `MPI_Bcast()`. Η διεργασία με ταυτότητα `rootrank` είναι αυτή που διαθέτει το δεδομένο και επομένως, στη δική της κλήση

βάζει έναν δείκτη στον χώρο που βρίσκεται αποθηκευμένο το δεδομένο αυτό (buf). Οι υπόλοιπες διεργασίες, στο δικό τους buf υποδεικνύουν τον χώρο στον οποίο επιθυμούν να τοποθετηθεί το δεδομένο κατά τη λήψη.

Διασκόρπιση — Η διασκόρπιση είναι λίγο πιο περίπλοκη, και γίνεται με την εξής κλήση:

```
MPI_Scatter(sbuf, sn, stype, rbuf, rn, rtype, rootrank,
            MPI_COMM_WORLD);
```

Αρχικά πρέπει να πούμε ότι η διεργασία που αποστέλλει τα μηνύματα είναι αυτή με ταυτότητα rootrank. Τα τρία πρώτα ορίσματα είναι σημαντικά μόνο για αυτή τη διεργασία, και ουσιαστικά το MPI τα αγνοεί στις κλήσεις που κάνουν οι υπόλοιπες διεργασίες. Κάτι άλλο που πρέπει να προσεχθεί είναι ότι, για λόγους συμμετρίας, το MPI θεωρεί ότι τα μηνύματα αποστέλλονται σε όλες τις διεργασίες, *συμπεριλαμβανομένης και της ίδιας της* rootrank. Το λεπτό αυτό σημείο, αν προσέξετε, απεικονίζεται καθαρά στο Σχ. 5.6, όπου π.χ. η διεργασία Δ_0 στέλνει πάντα και στον εαυτό της[3]. Με βάση τις παρατηρήσεις αυτές, τα ορίσματα της MPI_Scatter() έχουν ως εξής:

- Το sn είναι το μέγεθος του μηνύματος (πλήθος στοιχείων) που θα σταλεί σε κάθε διεργασία και *πρέπει* να είναι ίσο με το rn που αναφέρεται στο πλήθος στοιχείων που θα παραλάβει κάθε διεργασία.
- Το stype είναι ο τύπος των στοιχείων και πρέπει να είναι ίδιος με τον rtype.
- Το sbuf περιέχει *όλα* τα μηνύματα που πρέπει να αποσταλούν. Αν υπάρχουν συνολικά (συμπεριλαμβανομένης και της rootrank) N διεργασίες, θα αποσταλούν N μηνύματα, με το καθένα να έχει sn στοιχεία. Επομένως, το sbuf πρέπει να είναι ένας πίνακας μεγέθους N*sn στοιχείων. Τα πρώτα sn στοιχεία αποτελούν το μήνυμα που θα δοθεί στη διεργασία 0, τα επόμενα sn στοιχεία είναι το μήνυμα για τη διεργασία 1, κ.ο.κ.
- Το rbuf είναι ο χώρος στον οποίο η καλούσα διεργασία θα παραλάβει το μήνυμα που της αναλογεί. Προφανώς, πρέπει να διαθέτει μέγεθος sn (= rn) στοιχείων τύπου rtype.

Συλλογή — Η συλλογή είναι το αντίστροφο της διασκόρπισης, με μία διεργασία-παραλήπτη και όλες να στέλνουν ένα μήνυμα προς αυτή. Στο MPI η κλήση είναι η παρακάτω:

```
MPI_Gather(sbuf, sn, stype, rbuf, rn, rtype, targetrank,
           MPI_COMM_WORLD);
```

Τα ορίσματα έχουν αντίστοιχη λογική με την MPI_Scatter(). Συγκεκριμένα, η διεργασία targetrank θα λάβει όλα τα μηνύματα στο χώρο rbuf. Κάθε μήνυμα θα έχει sn (= rn)

[3]Εξυπακούεται ότι αυτό συνήθως γίνεται μέσω άμεσης αντιγραφής μνήμης και όχι πραγματικού μηνύματος.

στοιχεία τύπου stype (= rtype) και επομένως, αν υπάρχουν N διεργασίες, ο χώρος rbuf θα καταληφθεί από N*sn στοιχεία συνολικά. Κάθε διεργασία, συμπεριλαμβανομένης και της targetrank τοποθετεί το μήνυμα που αποστέλλει στο sbuf.

Πολλαπλή εκπομπή — Κατά την πολλαπλή εκπομπή, όλες οι διεργασίες εκπέμπουν από ένα μήνυμα. Κοιτώντας στο Σχ. 5.6, προσέξτε ότι αυτό ισοδυναμεί με το να εκτελέσουν όλες οι διεργασίες την ίδια συλλογή (gather). Να γιατί το MPI ονομάζει την κλήση αυτή «allgather»:

```
MPI_Allgather(sbuf, sn, stype, rbuf, rn, rtype, MPI_COMM_WORLD);
```

Τα ορίσματα είναι ακριβώς ίδια με αυτά της MPI_Gather(). Η μόνη διαφορά είναι ότι πρέπει *όλες οι διεργασίες* να παρέχουν χώρο rbuf για να παραλάβουν όλα τα μηνύματα.

Λειτουργίες υποβίβασης — Η βασική κλήση για υλοποίηση υποβιβάσεων (reductions) στο MPI είναι η MPI_Reduce():

```
MPI_Reduce(sbuf, rbuf, n, dtype, op, targetrank, MPI_COMM_WORLD);
```

Το δεδομένο που συνεισφέρει η κάθε διεργασία βρίσκεται στο sbuf και είναι μεγέθους n στοιχείων, τύπου dtype. Τα δεδομένα συνδυάζονται με την πράξη op και το τελικό αποτέλεσμα (ίδιου μεγέθους και τύπου) καταλήγει στον χώρο rbuf της διεργασίας targetrank. Αν το πλήθος των στοιχείων n είναι μεγαλύτερο του 1, η πράξη γίνεται για κάθε στοιχείο ξεχωριστά.

Το MPI παρέχει μία πληθώρα έτοιμων λειτουργιών υποβίβασης, όπως αθροίσματος, γινομένου, εύρεσης μεγίστου, ελαχίστου κλπ. Η τιμή του op μπορεί να είναι μία εκ των: MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR. Τέλος, υπάρχουν και τα MPI_MAXLOC και MPI_MINLOC που μαζί με το μέγιστο ή ελάχιστο, αντίστοιχα, επιστρέφουν και την ταυτότητα της διεργασίας που το διαθέτει. Σημειώνουμε ότι το MPI παρέχει τη δυνατότητα να ορίσει ο προγραμματιστής νέες, δικές του λειτουργίες υποβίβασης, τις οποίες μπορεί να τις χρησιμοποιήσει μέσω της MPI_Reduce(), θέτοντας κατάλληλη τιμή στο op.

Τέλος, για λόγους πληρότητας, πρέπει να πούμε ότι το MPI παρέχει επιπλέον κλήσεις που σχετίζονται με υποβιβάσεις, η ανάλυση των οποίων ξεφεύγει από τους στόχους μας εδώ. Συγκεκριμένα, πρόκειται για τις συναρτήσεις:

- MPI_Allreduce(), όπου το αποτέλεσμα της υποβίβασης το λαμβάνουν όλες οι διεργασίες.
- MPI_Reduce_scatter(), όπου γίνεται υποβίβαση πολλαπλών στοιχείων, και στη συνέχεια διασκόρπιση των αποτελεσμάτων.
- MPI_Scan(), για προθεματικές λειτουργίες υποβίβασης (prefix reductions), όπως για παράδειγμα, προθεματικά αθροίσματα.

Άλλες συλλογικές επικοινωνίες — Κλείνοντας, σημειώνουμε ότι το MPI παρέχει και κάποιες παραλλαγές στις συλλογικές επικοινωνίες που είδαμε παραπάνω. Επιπλέον, διαθέτει κι άλλες λειτουργίες που μπορούν να φανούν χρήσιμες σε κάποιες εφαρμογές. Χαρακτηριστικά αναφέρουμε την κλήση `MPI_Alltoall()` όπου όλες οι διεργασίες κάνουν διασκόρπιση αλλά και την κλήση `MPI_Barrier()` που αποτελεί κλήση φραγής, δηλαδή συγχρονίζει όλες τις διεργασίες μεταξύ τους.

5.4.2 Υπολογισμός της σταθεράς $\pi = 3.14159...$ με συλλογικές επικοινωνίες

Όπου μπορεί να εφαρμοστεί, η χρήση συλλογικών επικοινωνιών έχει πολλές φορές ευεργετικά αποτελέσματα, σε σχέση με τις ιδιωτικές ανταλλαγές μηνυμάτων ανάμεσα σε ζεύγη διεργασιών. Οι απλές εφαρμογές, που είδαμε μέχρι στιγμής, αποτελούν χαρακτηριστικά παραδείγματα. Ανατρέξτε στην Ενότητα 5.3.1 όπου φτιάξαμε το πρώτο μας πρόγραμμα για τον υπολογισμό της σταθεράς π. Οι παρακάτω γραμμές προέρχονται από το πρόγραμμα του Σχ. 5.3 και αφορούν στην εισαγωγή από τον χρήστη του πλήθους των διαστημάτων, και στη συνέχεια τη μετάδοσή του σε όλες τις διεργασίες.

```
13    if (myid == 0) {
14      printf“(Enter number of divisions: “);
15      scanf("%d", &N);
16      for (i = 1; i < nproc; i++)
17        MPI_Send(&N, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
18    }
19    else
20      MPI_Recv(&N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

Στο σημείο αυτό μπορούμε να αναγνωρίσουμε ότι πρόκειται για μία λειτουργία εκπομπής από τη διεργασία 0. Αντί, λοιπόν, να στείλουμε ιδιωτικά μηνύματα, μπορούμε να δώσουμε εντολή στο MPI να αναλάβει εξ ολοκλήρου τη διεκπεραίωση της επικοινωνίας, κάνοντας χρήση της `MPI_Bcast()`. Ο νέος κώδικας θα είναι ο εξής:

```
13    if (myid == 0) {
14      printf“(Enter number of divisions: “);
15      scanf("%d", &N);
16    }
17    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Όλες οι διεργασίες καλούν την `MPI_Bcast()` και δηλώνουν ότι η διεργασία 0 κάνει την εκπομπή. Η τελευταία αποστέλλει την τιμή του `N`, ενώ όλες θα τη λάβουν στο χώρο που βρίσκεται η δική τους μεταβλητή `N`.

Ένα ακόμα σημείο όπου μπορούμε να αναγνωρίσουμε ότι υπάρχει κάποιου είδους συλλογική επικοινωνία, είναι το τελευταίο στάδιο, όπου η διεργασία 0 συγκεντρώνει όλα

τα επί μέρους αποτελέσματα από τις υπόλοιπες διεργασίες και υπολογίζει την τελική τιμή του π:

```
28  if (myid == 0) {
29    for (i = 1; i < nproc; i++) {
30      MPI_Recv(&temp, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
31      result += temp;
32    }
33    printf("pi = %lf", result);
34  }
35  else
36    MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

Αρχικά, αντιλαμβανόμαστε ότι πρόκειται για μια λειτουργία συλλογής, με τη διεργασία 0 να λαμβάνει ένα διαφορετικό μήνυμα από κάθε άλλη διεργασία. Συνεχίζοντας τη σκέψη μας, βλέπουμε ότι τα δεδομένα που λαμβάνει η διεργασία 0 απλά τα προσθέτει μεταξύ τους. Αυτή, όμως, είναι μία τυπική περίπτωση υποβίβασης αθροίσματος. Επομένως, μπορούμε να κάνουμε χρήση των υποβιβάσεων που παρέχει το MPI και να καταλήξουμε στον παρακάτω κώδικα:

```
28  MPI_Reduce(&result, &temp, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
29  if (myid == 0)
30    printf("pi = %lf", temp);
```

Όπως βλέπουμε, όλες οι διεργασίες θα συνεισφέρουν με το `result` τους. Το τελικό άθροισμα (λόγω του `MPI_SUM`) θα το λάβει η διεργασία 0 στο `temp`, το οποίο και εκτυπώνει ως αποτέλεσμα. Ο τελικός κώδικας δίνεται στο Σχ. 5.7.

5.4.3 Πίνακας επί διάνυσμα με συλλογικές επικοινωνίες

Θα δούμε πάλι τον υπολογισμό πίνακα επί διάνυσμα που σχεδιάσαμε στην Ενότητα 5.3.2, αυτή τη φορά κάνοντας χρήση των πιο πρόσφατων γνώσεών μας. Το βασικό βήμα είναι πάντα να ανακαλύψουμε ποια είναι τα σημεία τα οποία κρύβουν συλλογικές επικοινωνίες. Έχουμε ήδη πει ότι η αποστολή του διανύσματος `v[]` από τη διεργασία 0 είναι μία περίπτωση εκπομπής. Επίσης, με λίγη παρατήρηση, βλέπουμε ότι η αρχική αποστολή των γραμμών από τον πίνακα `A` στις διάφορες διεργασίες αποτελεί περίπτωση διασκόρπισης. Ομοίως, απαιτείται η λειτουργία της συλλογής προκειμένου η διεργασία 0 να λάβει από κάθε διεργασία τα υπολογισμένα στοιχεία, ώστε να σχηματιστεί το συνολικό αποτέλεσμα. Το τελικό πρόγραμμα, το οποίο κάνει χρήση εκπομπής, διασκόρπισης και συλλογής δίνεται στο Σχ. 5.8.

Το πρόγραμμα βελτιώθηκε από όλες τις απόψεις κατά πολύ. Προσέξτε μόνο κάποιες μικρές λεπτομέρειες στην υλοποίηση. Επειδή το `WORK` υπολογίζεται κατά την εκτέλεση (εξαρτάται από το `nproc`), δεσμεύουμε δυναμικά σε κάθε διεργασία τον απαραίτητο χώρο

```
1  #include <mpi.h>
2
3  int main(int argc, char *argv[]) {
4    double     W, result = 0.0, temp;
5    int        N, i, myid, nproc;
6    MPI_Status status;
7
8    MPI_Init(&argc, &argv);
9    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
10   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
11
12   /* Initialization */
13   if (myid == 0) {
14     printf("Enter number of divisions: ");
15     scanf("%d", &N);
16   }
17   MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
18
19   /* The actual computation */
20   W = 1.0 / N;
21   for (i = myid; i < N; i += nproc)
22     result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
23
24   /* Gather results */
25   MPI_Reduce(&result, &temp, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
26   if (myid == 0)
27     printf("pi = %f\n", temp);
28
29   MPI_Finalize();
30   return 0;
31 }
```

Πρόγραμμα 5.7 *Βελτιωμένος υπολογισμός της σταθεράς π. (mp-pi2-collective.c)*

για τις WORK γραμμές του B, μέσω μίας συνάρτησης allocmatrix(). Παρόμοια, δεσμεύεται ο απαιτούμενος χώρος όπου θα αποθηκευτούν τα WORK στοιχεία του αποτελέσματος που θα υπολογίσει η κάθε διεργασία (διάνυσμα mypart).

Στη συνέχεια, γίνεται η διασκόρπιση των γραμμών του A. Κάθε ένα από τα μηνύματα που διασκορπίζονται αποτελείται από WORK γραμμές του A (και άρα είναι μεγέθους WORK*N στοιχείων). Η εκπομπή που ακολουθεί έχει ως αποτέλεσμα τη λήψη του v από όλες τις διεργασίες. Τέλος, αφού ολοκληρωθούν οι υπολογισμοί, κάθε διεργασία στέλνει αυτά που υπολόγισε (mypart) στη διεργασία 0, πακεταρισμένα σε ένα μήνυμα μεγέθους WORK στοιχείων. Οι αποστολές και η λήψη γίνονται μέσω μίας κλήσης συλλογής (MPI_Gather(), στη γραμμή 24).

```
#define N 1000          /* Matrix dimension */

void matrix_times_vector(int myid, int nproc) {
  int        i, j;
  int        WORK = N / nproc;    /* Rows per process */
  double     sum, v[N], A[N][N], res[N], *mypart, (*B)[N];

  if (myid == 0)
    initialize_elements(A, v);

  B = allocrows(WORK);            /* Allocate space for my rows.. */
  mypart = allocvector(WORK);     /* ..and for my results */

  MPI_Scatter(A, WORK*N, MPI_DOUBLE, B, WORK*N, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
  MPI_Bcast(v, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  for (i = 0; i < WORK; i++) {
    for (sum = 0.0, j = 0; j < N; j++)
      sum += B[i][j]*v[j];
    mypart[i] = sum;
  }

  MPI_Gather(mypart, WORK, MPI_DOUBLE, res, WORK, MPI_DOUBLE, 0,
             MPI_COMM_WORLD);
  if (myid == 0)
    show_result(res);
}
```

Πρόγραμμα 5.8 *Τρίτη εκδοχή του υπολογισμού πίνακα επί διάνυσμα σε* MPI, *με χρήση συλλογικών επικοινωνιών. (mp-mxv3-collective.c)*

5.4.4 Αποτίμηση των συλλογικών επικοινωνιών

Οι συλλογικές επικοινωνίες αποτελούν μία σημαντική ευκολία που μπορεί να παρέχεται από το μοντέλο μεταβίβασης μηνυμάτων. Η καθολική επικράτηση του MPI στηρίχτηκε σίγουρα και στην ευρεία γκάμα συλλογικών επικοινωνιών που παρέχει στον προγραμματιστή.

Οι λόγοι που πρέπει κανείς να φροντίζει να κάνει χρήση των συλλογικών επικοινωνιών, όπου μπορεί και όπου παρέχονται, είναι πολλοί. Κατ' αρχήν, πρέπει να έχει γίνει εμφανές ότι η χρήση των κλήσεων αυτών οδηγεί σε *μείωση του κώδικα*, η οποία μερικές φορές είναι θεαματική. Δείτε πόσο πιο σύντομο είναι το πρόγραμμα στο Σχ. 5.8 σε σχέση με αυτό του Σχ. 5.5, παρότι λειτουργικά είναι ισοδύναμα. Ένα δεύτερο και πιο σημαντικό σημείο είναι ότι ταυτόχρονα υπάρχει και *βελτίωση του κώδικα*. Πρόκειται για κώδικα ο οποίος είναι πλέον πιο ευανάγνωστος, ενώ η λειτουργία του γίνεται ευκολότερα αντιλη-

πτή. Η διαχείριση και αποσφαλμάτωσή του είναι, έτσι, πιο απλές καθώς μία συλλογική επικοινωνία περιγράφει εύγλωττα τη συνολική λειτουργία που κάνει το πρόγραμμά μας στο σημείο εκείνο—κάτι τέτοιο δεν είναι πάντα εύκολο όταν τα μηνύματα ανταλλάσσονται ένα-ένα και πιθανώς σε διαφορετικές περιοχές του κώδικα.

Ένας ακόμα λόγος, ο οποίος από μερικούς θεωρείται και ο πιο ισχυρός, είναι οι ευεργετικές επιδράσεις που μπορεί να υπάρχουν στις *επιδόσεις* των προγραμμάτων. Ενημερώνοντας το σύστημα για το είδος των επικοινωνιών που πρόκειται να γίνουν, το αφήνουμε αυτό το ίδιο να φροντίσει να τις διεκπεραιώσει όσο καλύτερα μπορεί. Έχουμε ήδη πει ότι κάποιες συλλογικές επικοινωνίες, εφόσον γίνουν επάνω από δίκτυα διασύνδεσης που το επιτρέπουν, μπορούν να ολοκληρωθούν πολύ ταχύτερα από το να στέλναμε, ως προγραμματιστές, ένα-ένα ξεχωριστά τα μηνύματα. Υπάρχουν υλοποιήσεις του MPI που εκμεταλλεύονται το δίκτυο που παρέχει το σύστημα για το οποίο σχεδιάστηκαν, και πραγματοποιούν πολλές από τις συλλογικές επικοινωνίες με αλγορίθμους εξαιρετικά υψηλών επιδόσεων.

5.5 Χρονική επικάλυψη υπολογισμών και επικοινωνιών

Το βασικότερο μέλημα κατά τη βελτιστοποίηση των επιδόσεων ενός προγράμματος, το οποίο στηρίζεται στη μεταβίβαση μηνυμάτων, είναι η μείωση του χρόνου που σπαταλάται σε επικοινωνίες. Προκειμένου να γίνει αυτό, εφαρμόζουμε τεχνικές σαν αυτές που είδαμε στις προηγούμενες ενότητες, όπως για παράδειγμα μείωση του πλήθους των μικρών μηνυμάτων ή χρήση συλλογικών επικοινωνιών, με κατάλληλη αναδιοργάνωση του κώδικά μας. Πολλές φορές, όμως, αυτό δεν είναι αρκετό· ο χρόνος που χάνεται, κυρίως στην αναμονή για παραλαβή μηνυμάτων που δεν έχουν καταφτάσει εγκαίρως, είναι σημαντικός.

Στις περιπτώσεις αυτές, και εφόσον η δομή του υπολογισμού το επιτρέπει, πιθανώς μπορεί να εφαρμοστεί η εξής ιδέα: όσο ο παραλήπτης περιμένει να του έρθει ένα απαραίτητο μήνυμα για να συνεχίσει την εργασία του, θα μπορούσε να εκτελεί άλλους χρήσιμους υπολογισμούς, που θα τους έκανε ούτως ή άλλως αργότερα. Μόλις καταφτάσει το μήνυμα, μπορεί να συνεχίσει την αρχική του εργασία. Με αυτόν τον τρόπο καταφέρνει να μειώσει (ίσως και να μηδενίσει) το χρόνο που θα καθόταν μπλοκαρισμένος, μέχρι να παραλάβει το μήνυμα που περιμένει. Με άλλα λόγια, *επικαλύπτει χρονικά τον χρόνο επικοινωνιακής αναμονής με χρήσιμους υπολογισμούς*. Η προχωρημένη αυτή τεχνική, εφόσον μπορεί να εφαρμοστεί, οδηγεί συχνά σε δραστική μείωση του κόστους των επικοινωνιών.

Πώς θα μπορούσε να υλοποιηθεί κάτι τέτοιο όμως; Σε γενικές γραμμές, υπάρχουν δύο κύριοι τρόποι. Ο πρώτος και μάλλον πιο πολύπλοκος τρόπος είναι να οργανώσουμε κάθε διεργασία, ώστε να αποτελείται από δύο νήματα. Η διαδικασία θα πρέπει να μας είναι γνωστή από το Κεφ. 4. Το ένα από τα νήματα θα αναλάβει μόνο τις επικοινωνίες,

ενώ το δεύτερο μόνο τους υπολογισμούς. Έτσι, όσο το ένα νήμα περιμένει μπλοκαρισμένο να ολοκληρωθεί μία λήψη, το άλλο θα μπορούσε να εκτελείται από τον επεξεργαστή του κόμβου όπου βρίσκεται η διεργασία, και να κάνει κάποιους υπολογισμούς. Φυσικά, τα δύο νήματα θα πρέπει να συντονίζονται σε συγκεκριμένα σημεία, ώστε το δεύτερο να λαμβάνει κατάλληλα τα δεδομένα που απαιτούνται για τους υπολογισμούς του. Η λύση είναι αρκετά γενική αλλά και αρκετά επίπονη στην υλοποίησή της. Επίσης, απαιτεί πολύ προσεκτική δρομολόγηση των νημάτων προκειμένου να βελτιωθούν οι επιδόσεις, χωρίς βέβαια να εξασφαλίζεται ότι θα πετυχαίνουμε πάντα αυτά που περιμένουμε.

Ένας δεύτερος τρόπος, για να γίνει χρονική επικάλυψη επικοινωνιών και υπολογισμών, είναι να παρέχει το προγραμματιστικό μοντέλο την ευκολία αυτή, μέσω ειδικών κλήσεων. Το MPI, συγκεκριμένα, διαθέτει και αυτή τη δυνατότητα, μέσω των λεγόμενων *μη-εμποδιστικών* (non-blocking) λήψεων και αποστολών. Η `MPI_Recv()` που χρησιμοποιείται από τον παραλήπτη για τη λήψη ενός μηνύματος είναι *εμποδιστική* (blocking) κλήση διότι ο παραλήπτης μπλοκάρει, μέχρι να έρθει το μήνυμα που περιμένει. Επομένως, δεν μπορεί να χρησιμοποιηθεί για να πετύχουμε τη χρονική επικάλυψη που επιθυμούμε. Προκειμένου να το πετύχουμε, θα πρέπει να χρησιμοποιηθεί η συνάρτηση `MPI_Irecv()` η οποία *ξεκινά* τη διαδικασία της λήψης αλλά *επιστρέφει αμέσως*. Αν το μήνυμα έχει ήδη φτάσει, η συνάρτηση αυτή δουλεύει ακριβώς όπως η κλασική `MPI_Recv()` και παραλαμβάνει το μήνυμα. Αν, όμως, το μήνυμα δεν έχει φτάσει, επιστρέφει αμέσως (χωρίς να παραλάβει κάτι) και η διεργασία μπορεί να συνεχίσει την εκτέλεσή της, κάνοντας πιθανώς κάτι άλλο. Η κλήση έχει ακριβώς τις ίδιες παραμέτρους με την `MPI_Recv()`, πλην της τελευταίας:

```
MPI_Irecv(buf, n, dtype, fromrank, tag, MPI_COMM_WORLD, req);
```

όπου το `req` είναι τύπου δείκτη προς `MPI_Request`, και αποτελεί ένα είδος αποδεικτικού για την αίτηση λήψης που γίνεται. Το αποδεικτικό αυτό χρησιμοποιείται σε μετέπειτα κλήσεις ελέγχου, για να διαπιστώνεται η κατάσταση της λήψης. Όπως είπαμε, η `MPI_Irecv()` επιστρέφει αμέσως μετά την κλήση της είτε παρέλαβε το μήνυμα είτε όχι. Προκειμένου να δούμε αν *ολοκληρώθηκε η λήψη*, θα πρέπει να ελέγξουμε την κατάστασή της, καλώντας τη συνάρτηση:

```
MPI_Test(req, flag, status);
```

όπου:

- `req` είναι το τελευταίο όρισμα της `MPI_Irecv()`.
- Το `flag` δείχνει σε έναν ακέραιο, στον οποίο θα επιστραφεί 1 (δηλαδή αληθές), αν το μήνυμα όντως παραλήφθηκε, και 0 (ψευδές), αν όχι.
- Το `status` είναι ίδιο με το τελευταίο όρισμα της `MPI_Recv`, και περιέχει πληροφορίες για το μήνυμα, εφόσον παραλήφθηκε, όπως π.χ. τον αποστολέα, την ετικέτα κλπ.

Εάν το μήνυμα δεν έχει παραληφθεί, είναι προφανές ότι ο έλεγχος πρέπει να μπει μέσα σε κάποιον βρόχο, όπου εξετάζουμε για ολοκλήρωση της λήψης, επαναληπτικά. Εφόσον δεν υπάρχει κάτι άλλο να κάνει η διεργασία, στο τέλος θα πρέπει να εκτελεί συνεχώς τον βρόχο αυτό, μέχρι να επιστραφεί 1 στο flag. Στην περίπτωση αυτή, μία άλλη επιλογή είναι να μπλοκάρουμε ρητά τη διεργασία με την εξής κλήση,

```
MPI_Wait(req, status);
```

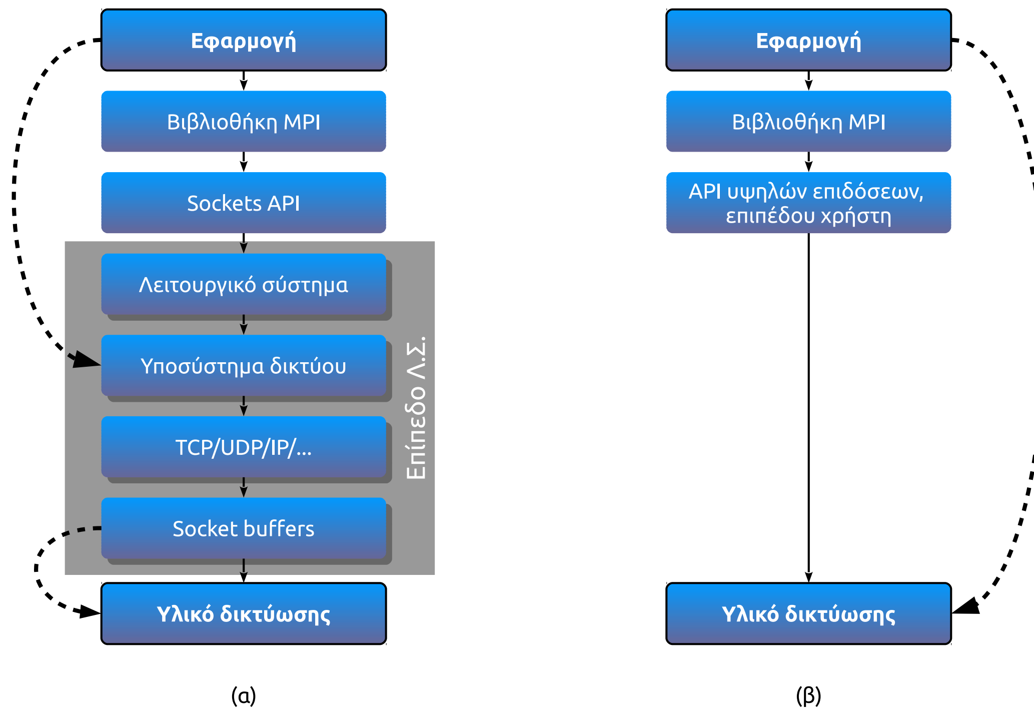
η οποία, επιστρέφει μόνο όταν έρθει το μήνυμα.[4]

Πριν δούμε ένα παράδειγμα όπου φαίνεται η χρήση των παραπάνω κλήσεων, πρέπει να επισημάνουμε ότι υπάρχουν και κλήσεις *μη εμποδιστικής αποστολής*. Τι ακριβώς, όμως, σημαίνει «μη εμποδιστική» αποστολή; Υποθέτοντας, πάντα, το ασύγχρονο μοντέλο επικοινωνίας μεταξύ διεργασιών (Σχ. 5.2), η κλήση αποστολής δεν επιστρέφει αμέσως; Η απάντηση είναι, περιέργως, αρνητική. Ναι μεν δεν περιμένει τον παραλήπτη να παραλάβει το μήνυμα, όμως *περιμένει να μεταφερθεί το μήνυμα στο δίκτυο*. Και μάλιστα, η αναμονή αυτή δεν είναι καθόλου αμελητέα στα σύγχρονα υπολογιστικά συστήματα.

Για να γίνει κατανοητή η έννοια της εμποδιστικότητας κατά την αποστολή μηνύματος, πρέπει να δούμε με λεπτομέρεια πώς ακριβώς υλοποιείται σε ένα τυπικό σύστημα. Στο Σχ. 5.9(α) φαίνεται η πλήρης διαδικασία που ακολουθείται από τη στιγμή που η εφαρμογή κάνει κλήση στην MPI_Send(), μέχρι τη στιγμή που τα bits του μηνύματος τοποθετούνται πάνω στο καλώδιο και μεταφέρονται προς τον προορισμό. Η βιβλιοθήκη του MPI, η οποία υλοποιεί την MPI_Send(), χρησιμοποιεί τις κλήσεις των υποδοχών (sockets) που παρέχονται από το σύστημα. Οι κλήσεις αυτές προκαλούν μετάβαση από το επίπεδο του χρήστη, στο επίπεδο του πυρήνα του λειτουργικού συστήματος. Εκεί, το τμήμα το οποίο ασχολείται με τη δικτύωση κάνει χρήση των κατάλληλων πρωτοκόλλων, προκειμένου να μετατρέψει το μήνυμα του χρήστη σε μία σειρά από bytes τα οποία είναι κατάλληλα για μεταφορά επάνω στο δίκτυο, προσθέτοντας επικεφαλίδες, κώδικες διόρθωσης λαθών κλπ. Τα προς αποστολή bytes φυλάσσονται στους socket buffers, από όπου τελικά μεταφέρονται στην κάρτα δικτύου, η οποία με τη σειρά της τα τοποθετεί επάνω στο φυσικό μέσο δικτύωσης.

Στο Σχ. 5.9(α), με διακεκομμένα βέλη, απεικονίζονται οι πιθανότερες αντιγραφές δεδομένων. Πρόκειται για μεταφορές ή αντιγραφές από ένα σημείο της κύριας μνήμης σε ένα άλλο (memory copies) που απαιτούνται από τους αλγορίθμους διαχείρισης της δικτύωσης. Συγκεκριμένα, υποθέτοντας ότι η ίδια η βιβλιοθήκη του MPI δεν προκαλεί αντιγραφές (δηλαδή χειρίζεται μόνο δείκτες στα δεδομένα), μία αντιγραφή σίγουρα θα συμβεί, όταν από το επίπεδο του χρήστη μεταβούμε σε επίπεδο πυρήνα λειτουργικού συστήματος, καθώς ο πυρήνας διαθέτει δικό του χώρο διευθύνσεων για τις εσωτερικές του δομές. Στη συνέχεια, μπορεί να υπάρξουν επιπλέον αντιγραφές, αλλά κατ' ελάχιστον θα πρέπει να

[4]Ουσιαστικά, η MPI_Recv() ισοδυναμεί με κλήση στην MPI_Irecv(), ακολουθούμενη αμέσως από κλήση στην MPI_Wait().



Σχήμα 5.9 *Η διαδικασία αποστολής μηνύματος: (α) σε μία τυπική υλοποίηση του* MPI *σε συνήθη δίκτυα όπως το ethernet και (β) υλοποίηση υψηλών επιδόσεων σε γρήγορα δίκτυα. Τα διακεκομμένα βέλη αντιπροσωπεύουν πιθανές αντιγραφές δεδομένων.*

αντιγραφούν τα τελικά bytes από τους buffers του πυρήνα στους buffers της κάρτας δικτύωσης. Με άλλα λόγια, τα δεδομένα του μηνύματος θα αντιγραφούν τουλάχιστον δύο φορές, και πολλές φορές, ανάλογα με την υλοποίηση και την κατάσταση του συστήματος, παραπάνω από δύο φορές, πριν καταλήξουν στο καλώδιο (αντίστοιχες διαδικασίες συμβαίνουν και στη μεριά του παραλήπτη). Στα σύγχρονα συστήματα, η ταχύτητα των δικτύων είναι τόσο μεγάλη ώστε, πλέον, *οι αντιγραφές στη μνήμη μπορεί να είναι πιο χρονοβόρες από τις αντίστοιχες μεταφορές πάνω στο δίκτυο*. Το τελικό συμπέρασμα είναι ότι κατά την αποστολή του μηνύματος, το κομμάτι εκείνο που αφορά στα τεκταινόμενα στον κόμβο του αποστολέα, αποτελεί πολύ σημαντική αιτία καθυστερήσεων.

Για λόγους πληρότητας επισημαίνουμε ότι, σε προχωρημένα συστήματα, έχουν αναπτυχθεί τεχνικές που υλοποιούν την αποστολή των μηνυμάτων εξαιρετικά γρήγορα. Συγκεκριμένα, όπως φαίνεται και στο Σχ. 5.9(β), γίνεται πλήρης παράκαμψη του λειτουργικού συστήματος, και από το επίπεδο του χρήστη πραγματοποιείται άμεση πρόσβαση στο υλικό του δικτύου. Με αυτόν τον τρόπο, όχι μόνο αποφεύγεται η σημαντική καθυστέρηση μετάβασης στο επίπεδο του πυρήνα, μειώνεται επίσης και το πλήθος των αντιγραφών που πρέπει να γίνουν. Οι τεχνικές αυτές, συνδυαζόμενες με υποδομή δικτύου υψηλών επιδόσεων / χαμηλών καθυστερήσεων, οδηγούν στις λεγόμενες υλοποιήσεις *μηδενικών αντιγραφών* (zero copy), όπου αποφεύγονται εντελώς οι ενδιάμεσες αντιγραφές των δεδομένων, και με κατάλληλη διαχείριση του ελεγκτή δικτύου, τα bytes «ρέουν» από τον χώρο της εφαρμογής

κατευθείαν επάνω στο καλώδιο. Πρέπει να είναι κατανοητό ότι η όλη διαδικασία ολοκληρώνεται πολύ πιο γρήγορα από αυτή του Σχ. 5.9(α), χωρίς, βέβαια, να γίνεται ποτέ εντελώς αμελητέα.

Επανερχόμενοι, λοιπόν, στο ερώτημα της εμποδιστικότητας των αποστολών, αρκεί να πούμε ότι η κλήση αποστολής επιστρέφει όταν ολοκληρωθεί όλη η διαδικασία του Σχ. 5.9(α) και το τελευταίο bit φύγει επάνω στο καλώδιο. Ο αποστολέας, ναι μεν δεν περιμένει τον παραλήπτη να παραλάβει το μήνυμα, μπλοκάρει, όμως, μέχρι το σύστημα να «αδειάσει» πλήρως όλα τα bits στο δίκτυο. Και επειδή, όπως είπαμε, ο χρόνος αυτός δεν είναι διόλου αμελητέος, το MPI παρέχει τη μη-εμποδιστική κλήση:

```
MPI_Isend(buf, n, dtype, torank, tag, MPI_COMM_WORLD, req);
```

όπου το req είναι το «αποδεικτικό» (δείκτης σε MPI_Request), όπως ακριβώς και στην κλήση της MPI_Irecv(). Η συνάρτηση αυτή απλά *ξεκινά* τη διαδικασία αποστολής και επιστρέφει αμέσως. Με βάση το Σχ. 5.9(α), θα λέγαμε ότι επιστρέφει πριν καν η βιβλιοθήκη του MPI κάνει χρήση των κλήσεων της διεπαφής (API) των sockets. Καθώς το σύστημα ολοκληρώνει τη διαδικασία της αποστολής, η καλούσα διεργασία είναι ελεύθερη να εκτελεί ταυτόχρονα χρήσιμο κώδικα, επικαλύπτοντας έτσι χρονικά επικοινωνία και υπολογισμούς.

Στις περιπτώσεις που, μετά τη μη-εμποδιστική κλήση αποστολής, ακολουθεί αποστολή νέου μηνύματος, διατρέχουμε τον κίνδυνο να μην έχει ολοκληρωθεί η προηγούμενη αποστολή, δηλαδή να μην έχουν προωθηθεί όλα τα bytes του προηγούμενου μηνύματος στο δίκτυο. Αν αυτό συμβεί, τότε οι συνέπειες μπορεί να είναι καταστροφικές, καθώς υπάρχει περίπτωση να προλάβουν να «πανωγραφούν» ανεπανόρθωτα κάποια δεδομένα, πριν αυτά μεταφερθούν με ασφάλεια. Προκειμένου να δούμε αν *ολοκληρώθηκε η μη εμποδιστική αποστολή*, θα πρέπει να ελέγξουμε την κατάστασή της, καλώντας και εδώ τη συνάρτηση

```
MPI_Test(req, flag, status);
```

όπου, όπως και στην περίπτωση της μη εμποδιστικής λήψης, στο flag θα επιστραφεί 1 (αληθές) αν το μήνυμα όντως έφυγε πλήρως από τον κόμβο, και 0 (ψευδές) αν όχι. Η διεργασία μπορεί να μπλοκάρει ρητά μέχρι την ολοκλήρωση της αποστολής με κλήση στην MPI_Wait(), ακριβώς όπως και στη μη εμποδιστική λήψη.[5]

Ολοκληρώνουμε την ενότητα αυτή με ένα μικρό παράδειγμα κώδικα που κάνει χρήση μη εμποδιστικών επικοινωνιών. Κάθε διεργασία του προγράμματος υποτίθεται ότι καλεί τη συνάρτηση που ακολουθεί:

```
void workloop(int previd, int nextid)
{
  MPI_Status status;
  double     v[N];

  while ( check_termination() ) {
```

[5] Η MPI_Isend(), ακολουθούμενη αμέσως από κλήση στην MPI_Wait(), ισοδυναμεί με την MPI_Send().

```
    MPI_Recv(v, N, MPI_DOUBLE, previd, 0, MPI_COMM_WORLD, &status);
    do_calculation(v, N);
    MPI_Send(v, N, MPI_DOUBLE, nextid, 0, MPI_COMM_WORLD);
  }
}
```

και εκτελεί επαναληπτικά το εξής σενάριο: λαμβάνει ένα δεδομένο (διάνυσμα v, με N στοιχεία) από την προηγούμενη διεργασία (previd), κάνει έναν χρονοβόρο υπολογισμό και τροποποιεί το δεδομένο αυτό (do_calculation()), και στη συνέχεια το μεταβιβάζει στην επόμενη διεργασία (nextid). Το σενάριο αυτό αποτελεί βασική λειτουργία εφαρμογών που στηρίζονται στην τεχνική της διοχέτευσης (pipelining) των υπολογισμών.

Ο χρόνος που αφιερώνεται σε κάθε επανάληψη, ισούται με το άθροισμα της αναμονής στην MPI_Recv() συν τον χρόνο των υπολογισμών συν την αναμονή στην MPI_Send(). Με τη

```
1  void workloop(int previd, int nextid) {
2    MPI_Status  status;
3    MPI_Request rreq, sreq;
4    double      v1[N], v2[N], *v, *nextv, *tmp;
5    int         firstiter = 1;
6
7    v = v1;
8    nextv = v2;
9    MPI_Irecv(v, N, MPI_DOUBLE, previd, 0, MPI_COMM_WORLD, &rreq);
10
11   while ( check_termination() ) {
12     MPI_Wait(&rreq, &status);   /* Wait for last Irecv to complete */
13
14     /* Start reception for *next* iteration */
15     MPI_Irecv(nextv, N, MPI_DOUBLE, previd, 0, MPI_COMM_WORLD, &rreq);
16
17     do_calculation(v);          /* Work on the received vector */
18
19     if (!firstiter)
20       MPI_Wait(&sreq, &status); /* Wait for last Isend to complete */
21     else
22       firstiter = 0;
23
24     MPI_Isend(v, N, MPI_DOUBLE, nextid, 0, MPI_COMM_WORLD, &sreq);
25
26     tmp = v; v = nextv; nextv = tmp;           /* Swap v and nextv */
27   }
28 }
```

Πρόγραμμα 5.10 *Παράδειγμα χρήσης μη εμποδιστικών επικοινωνιών (το οποίο, όμως, έχει ένα μικρό πρόβλημα). (mp-nonblock.c)*

χρήση μη εμποδιστικών επικοινωνιών, μπορεί, υπό ευνοϊκές συνθήκες φυσικά, να εξαλειφθεί η αναμονή για τις επικοινωνίες ή τουλάχιστον να μειωθεί στο ελάχιστο δυνατό, και σε κάθε επανάληψη ο χρόνος να καταναλώνεται σχεδόν αποκλειστικά στους υπολογισμούς. Μία μη εμποδιστική εκδοχή δίνεται στο Σχ. 5.10.

Για να μπορούμε να κάνουμε υπολογισμούς ταυτόχρονα με την εξέλιξη μίας λήψης ή μίας αποστολής, θα πρέπει να χρησιμοποιήσουμε δύο διανύσματα αντί για ένα, πράγμα που σημαίνει διπλάσιο χώρο (η τεχνική είναι γνωστή και ως *double-buffering*). Όσο λαμβάνουμε το επόμενο διάνυσμα, ο υπολογισμός γίνεται επάνω στο προηγούμενο που λάβαμε. Τα δύο διανύσματα είναι τα `v1[]` και `v2[]`. Ας υποθέσουμε ότι έχουμε λάβει ήδη στο `v1`, και ότι έχουμε ξεκινήσει να λαμβάνουμε (μη εμποδιστικά) το επόμενο διάνυσμα στον χώρο του `v2`. Όταν τελειώσει ο υπολογισμός της `do_calculation()`, θα πρέπει να στείλουμε το τροποποιημένο `v1` στην επόμενη διεργασία και να πάμε στην επόμενη επανάληψη. Εκεί, οι ρόλοι αλλάζουν. Θα πρέπει να κάνουμε υπολογισμούς με το `v2[]`, το οποίο, στο μεταξύ, ελπίζουμε ότι θα έχει καταφτάσει, ενώ θα ξεκινήσουμε τη λήψη του αμέσως επόμενου διανύσματος στον χώρο του `v1`. Για ευκολία, αντί να κάνουμε αυτή την αλλαγή ρόλων ανάμεσα στα δύο διανύσματα, από επανάληψη σε επανάληψη, χρησιμοποιούμε δύο δείκτες `v` και `nextv`. Ο πρώτος δείχνει στο διάνυσμα στο οποίο θα κάνουμε τους υπολογισμούς και ο δεύτερος στο διάνυσμα που περιμένουμε να λάβουμε για την επόμενη επανάληψη. Στο τέλος κάθε επανάληψης κάνουμε αλλαγή των ρόλων με απλή εναλλαγή των δύο δεικτών (γραμμή 26).

Προκειμένου να σιγουρευτούμε ότι έχει καταφτάσει το διάνυσμα που περιμένουμε για την τρέχουσα επανάληψη (το `v[]`, όπως είπαμε), μπλοκάρουμε μέχρι να ολοκληρωθεί η τελευταία λήψη που κάναμε, με χρήση της `MPI_Wait()`, στη γραμμή 11. Στη συνέχεια, ξεκινάμε αμέσως μη εμποδιστική λήψη για το επόμενο διάνυσμα (`nextv`), στη γραμμή 14. Όσο εξελίσσεται η λήψη αυτή, η `MPI_Irecv()` έχει επιστρέψει και εμείς εκτελούμε ήδη τους υπολογισμούς στη γραμμή 16. Τέλος, στέλνουμε το αποτέλεσμα στην επόμενη διεργασία (γραμμή 24). Επειδή όμως, η αποστολή είναι μη εμποδιστική, πρέπει να μπλοκάρουμε μέχρι να ολοκληρωθεί η προηγούμενη αποστολή που κάναμε. Αυτό γίνεται με χρήση μίας δεύτερης `MPI_Wait()`, στις γραμμές 19–22, σε όλες τις επαναλήψεις, πλην της πρώτης (διότι σε αυτήν δεν έχει προηγηθεί άλλη αποστολή).

Παρά το γεγονός ότι το πρόγραμμα στο Σχ. 5.10 θα δουλεύει, σε γενικές γραμμές, σωστά, υπάρχει μία λεπτομέρεια που μας έχει διαφύγει, και η οποία μπορεί θεωρητικά να οδηγήσει σε λάθος αποτελέσματα! Στους κώδικες που συνοδεύουν το βιβλίο αυτό περιέχεται η σωστή, ολοκληρωμένη εκδοχή. Αξίζει, όμως, να προσπαθήσετε να λύσετε το Πρόβλημα 5.6 πριν ανατρέξετε σε αυτούς. Κλείνοντας, είναι χρήσιμο ο αναγνώστης να θυμάται τα παρακάτω σημεία:

- Μήνυμα που στάλθηκε με μη εμποδιστική αποστολή μπορεί να παραληφθεί και με εμποδιστική λήψη,

- Μήνυμα που στάλθηκε με εμποδιστική αποστολή μπορεί να παραληφθεί και με μη εμποδιστική λήψη,
- Για τη χρήση μη εμποδιστικών επικοινωνιών, ο «χρυσός κανόνας» είναι να τις *ξεκινάμε νωρίς* και να τις *ολοκληρώνουμε αργά*.

5.6 Άλλα χαρακτηριστικά του MPI

Στις προηγούμενες ενότητες καλύψαμε τις πιο σημαντικές πλευρές του μοντέλου μεταβίβασης μηνυμάτων και είδαμε με λεπτομέρεια πώς αυτές υλοποιούνται στο MPI. Πρέπει να έχει γίνει σαφές ότι το MPI, εκτός από τις απλές δομές, παρέχει πληθώρα παραλλαγών και επιπρόσθετων δυνατοτήτων. Ο στόχος του κεφαλαίου αυτού είναι η μύηση στις τεχνικές που σχετίζονται με τη μεταβίβαση μηνυμάτων, γενικά ως μοντέλο, και όχι να αποτελέσει έναν πλήρη οδηγό αναφοράς για το MPI. Παρ' όλα αυτά, θεωρούμε σκόπιμο να κάνουμε μία εισαγωγή σε κάποια επιπλέον χαρακτηριστικά του MPI, η οποία μπορεί να αποτελέσει εφαλτήριο για τον αναγνώστη που ενδιαφέρεται να εμβαθύνει περισσότερο στα προχωρημένα θέματα του ισχυρού αυτού προτύπου.

Ομάδες διεργασιών, communicators και τοπολογίες — Το MPI δίνει τη δυνατότητα να οριστούν ομάδες (groups) από διεργασίες. Στην ορολογία του MPI μία ομάδα αντιπροσωπεύεται βασικά από έναν *communicator*. Όσες διεργασίες βρίσκονται στην ίδια ομάδα μπορούν να κάνουν κάποιες τοπικές λειτουργίες, εσωτερικά στην ομάδα αυτή, χωρίς να επηρεάζουν τις υπόλοιπες διεργασίες, χρησιμοποιώντας τον communicator της ομάδας. Για παράδειγμα, μπορούν να γίνονται συλλογικές επικοινωνίες όπου εμπλέκονται μόνο οι διεργασίες της συγκεκριμένης ομάδας.

Μία διεργασία μπορεί να ανήκει σε πολλές ομάδες. Σε κάθε ομάδα που ανήκει, έχει και διαφορετική ταυτότητα. Σε μία ομάδα με n διεργασίες, οι ταυτότητες (ranks) έχουν πάντα τιμές από 0 μέχρι $n-1$. Υποχρεωτικά, πάντως, κάθε διεργασία ανήκει στην *καθολική* ή *«παγκόσμια»* ομάδα, η οποία περιλαμβάνει *όλες τις διεργασίες*, έτσι όπως δημιουργήθηκαν από το `mpirun` στην αρχή της εκτέλεσης του προγράμματος. Η ομάδα αυτή αντιπροσωπεύεται από τον γνωστό μας communicator `MPI_COMM_WORLD`, ο οποίος είναι και ο μοναδικός που έχουμε χρησιμοποιήσει μέχρι τώρα. Σε κάθε επικοινωνία που γίνεται πρέπει να προσδιορίζεται η ομάδα στην οποία ανήκουν οι διεργασίες που επικοινωνούν. Έτσι εξηγείται το γεγονός ότι όλες οι κλήσεις του MPI που είδαμε, απαιτούν το `MPI_COMM_WORLD` ως όρισμα. Αν οι διεργασίες επικοινωνούν μέσω άλλης ομάδας, στη θέση του `MPI_COMM_WORLD`, πρέπει να μπει ο αντίστοιχος communicator.

Για να φτιάξουμε μία νέα ομάδα, συνήθως δημιουργούμε έναν νέο communicator. Ένας τρόπος να γίνει αυτό είναι να διαχωρίσουμε κάποιες διεργασίες από έναν υπάρχοντα

communicator, με την κλήση `MPI_Comm_split()`. Η κλήση αυτή είναι συλλογική και θα πρέπει να την κάνουν όλες οι διεργασίες της προϋπάρχουσας ομάδας. Μετά την κλήση, μπορούν να προκύψουν ένας ή παραπάνω νέοι communicators.

Ένας δεύτερος τρόπος είναι να ορίσουμε μία εικονική *τοπολογία* διεργασιών, η οποία προκαλεί και τη δημιουργία ενός νέου communicator. Οι τοπολογίες προσφέρουν έναν εναλλακτικό τρόπο στην αρίθμηση των διεργασιών· για παράδειγμα, αντί να έχουν ακολουθιακές ταυτότητες 0, 1, 2, ..., μπορούν να αριθμούνται σαν να βρίσκονται σε ένα διδιάστατο πλέγμα. Η διεργασία (i, j) θα είναι αυτή που βρίσκεται στη γραμμή i και στη στήλη j αυτού του εικονικού πλέγματος. Προσέξτε ότι είναι καθαρά και μόνο θέμα αρίθμησης, και ότι δεν υπάρχει κάποια τοπολογική διασύνδεση μεταξύ των διεργασιών. Ο λόγος να χρησιμοποιήσει κανείς τοπολογίες είναι απλά η διευκόλυνση που παρέχουν στην υλοποίηση κάποιων αλγορίθμων που έχουν σχεδιαστεί να λειτουργούν σε συγκεκριμένα δίκτυα διασύνδεσης. Η πιο συνηθισμένη κλήση για δημιουργία νέας τοπολογίας (και επομένως, νέας ομάδας και νέου communicator) είναι η κλήση `MPI_Cart_create()` που ορίζει πολυδιάστατα πλέγματα.

Γιατί, όμως, να χρησιμοποιήσει κάποιος ομάδες διεργασιών; Μία απάντηση είναι ότι υπάρχουν εφαρμογές, που διαθέτουν μία ιεραρχία στον παραλληλισμό που μπορεί να επιτευχθεί. Επιθυμούμε να μοιράσουμε τμήματα του υπολογισμού σε ανεξάρτητες ομάδες διεργασιών και στο τέλος να συγκεντρώσουμε τα αποτελέσματά τους. Σε κάθε ομάδα οι υπολογισμοί γίνονται παράλληλα, και όλες οι ομάδες εργάζονται ταυτόχρονα. Ένας δεύτερος λόγος είναι ότι, αν δεν υπήρχαν ομάδες, θα ήταν σχεδόν αδύνατον να υπάρξουν βιβλιοθήκες με έτοιμες υλοποιήσεις για βασικούς και κεντρικούς αλγορίθμους. Φανταστείτε, για παράδειγμα, ότι μέσα από την εφαρμογή μας καλούμε μία έτοιμη βιβλιοθήκη, η οποία υλοποιεί (παραλληλοποιημένο) έναν απαραίτητο υπολογισμό. Η βιβλιοθήκη χρησιμοποιεί το MPI για να μεταβιβάζει τα δικά της μηνύματα, τη στιγμή που η εφαρμογή στέλνει και λαμβάνει άλλα μηνύματα. Υποθέτοντας ότι και οι δύο έχουν κάνει αίτηση λήψης, όταν καταφτάσει ένα μήνυμα σε μία διεργασία, το MPI αδυνατεί να αποφασίσει αν θα το παραδώσει στη βιβλιοθήκη ή στον κώδικα της εφαρμογής. Η χρήση διαφορετικών ετικετών για τα μηνύματα δεν αποτελεί λύση, διότι μπορεί να μην έχουμε πρόσβαση στον κώδικα της βιβλιοθήκης. Έτσι, δεν θα γνωρίζουμε ποιες ετικέτες χρησιμοποιεί η βιβλιοθήκη, ώστε να τις αποφύγουμε στον κώδικα της εφαρμογής μας. Οι communicators παρέχουν τη λύση στο πρόβλημα αυτό. Οι βιβλιοθήκες ορίζουν δικούς τους communicators και οργανώνουν τις επικοινωνίες τους επάνω σε αυτούς, οπότε δεν υπάρχει ποτέ περίπτωση να μπλεχτούν με τις υπόλοιπες επικοινωνίες που κάνει η εφαρμογή.

Είναι σκόπιμο να σημειώσουμε ότι το MPI παρέχει communicators οι οποίοι χρησιμοποιούνται για επικοινωνία εσωτερικά σε μία ομάδα (intra-communicators) και άλλους που χρησιμοποιούνται για την επικοινωνία μεταξύ διαφορετικών ομάδων (inter-communicators).

Δυναμική διαχείριση διεργασιών — Όπως είπαμε και στην αρχή του κεφαλαίου, οι περισσότερες εφαρμογές του MPI, στηρίζονται σε στατικά καθορισμένο πλήθος διεργασιών, το οποίο προσδιορίζεται στην αρχή της εκτέλεσης μέσω του `mpirun`. Το MPI-2 όμως, παρέχει τη δυνατότητα δυναμικής δημιουργίας νέων διεργασιών κατά τη διάρκεια της εκτέλεσης μίας εφαρμογής. Η αλήθεια είναι ότι η δυνατότητα αυτή δεν ήταν ποτέ ούτε ιδιαίτερα δημοφιλής ούτε καθολικά διαθέσιμη. Όμως, καλό είναι να γνωρίζουμε ότι υπάρχει, για την περίπτωση που απαιτείται από την εφαρμογή μας.

Η δημιουργία νέων διεργασιών γίνεται με την κλήση `MPI_Comm_spawn()`. Πρόκειται για συλλογική κλήση από μία προϋπάρχουσα ομάδα. Το αποτέλεσμα είναι η δημιουργία νέων διεργασιών οι οποίες ανήκουν στον δικό τους «κόσμο»—έχουν δηλαδή το δικό τους `MPI_COMM_WORLD`, το οποίο είναι ξεχωριστό από αυτό των υπόλοιπων διεργασιών. Προκειμένου να επιτευχθεί επικοινωνία με τις νέες διεργασίες, επιστρέφεται ταυτόχρονα και ένας νέος inter-communicator.

Μονόπλευρες επικοινωνίες — Η επικοινωνία μεταξύ δύο διεργασιών, όπως την έχουμε δει μέχρι στιγμής, θεωρείται *αμφίπλευρη* (two-sided) διότι, για να ολοκληρωθεί, πρέπει να ενεργήσουν και οι δύο εμπλεκόμενες διεργασίες· η μία διεργασία πρέπει να εκτελέσει αποστολή και η άλλη λήψη. Στη λεγόμενη *μονόπλευρη* επικοινωνία (one-sided coommunication), ενεργεί μόνο ένα από τα δύο μέρη είτε ο αποστολέας είτε ο παραλήπτης. Στην περίπτωση αυτή, εκείνος που εκτελεί την επικοινωνία πρέπει να προσδιορίσει όλες τις απαραίτητες παραμέτρους. Για παράδειγμα, στη μονόπλευρη αποστολή, η πηγή πρέπει να καθορίσει τον χώρο που βρίσκεται το μήνυμα που θα σταλθεί αλλά και τον χώρο (στον προορισμό) όπου θα αποθηκευτεί και η αποστολή θα γίνει χωρίς τη μεσολάβηση του παραλήπτη. Οι μονόπλευρες επικοινωνίες είναι ευρύτερα γνωστές με τον όρο *απομακρυσμένη προσπέλαση μνήμης* (Remote Memory Access, RMA), όπου μία διεργασία μπορεί να προσβεί, γράφοντας ή διαβάζοντας, άμεσα τη μνήμη μίας άλλης.

Το MPI πρώτη φορά παρείχε τη δυνατότητα μονόπλευρων επικοινωνιών στην έκδοση 2.0, ενώ η έκδοση 3.0 πρόσθεσε μερικές επιπλέον δυνατότητες. Προκειμένου μία διεργασία να χρησιμοποιήσει μονόπλευρες επικοινωνίες, πρέπει να ορίσει έναν χώρο στη μνήμη της ο οποίος θα είναι διαθέσιμος για απομακρυσμένη πρόσβαση από όλες τις άλλες. Ο χώρος αυτός ονομάζεται *παράθυρο μνήμης* και καθορίζεται με την κλήση `MPI_Win_create()`, η οποία είναι συλλογική κλήση στην ομάδα που ανήκει η διεργασία. Το MPI-3 επιπλέον διαθέτει την κλήση `MPI_Win_create_dynamic()`, η οποία δεν είναι συλλογική και επιτρέπει σε μία διεργασία δυναμικά να καταχωρεί έναν δικό της χώρο ως προσβάσιμο από τις υπόλοιπες. Από τη στιγμή που έχουν καθοριστεί τα παράθυρα μνήμης, η μονόπλευρη αποστολή γίνεται με τη συνάρτηση `MPI_Put()` και η μονόπλευρη λήψη με την `MPI_Get()`. Η χρήση απομακρυσμένης πρόσβασης μνήμης υπόκειται σε κάποιους περιορισμούς και απαιτεί κατάλληλες κλήσεις συγχρονισμού.

5.7 Ανακεφαλαίωση και βιβλιογραφικές σημειώσεις

Στο κεφάλαιο αυτό ασχοληθήκαμε με το μοντέλο μεταβίβασης μηνυμάτων, το οποίο είναι, εκ πρώτης όψεως, ένα πολύ απλό μοντέλο παράλληλου προγραμματισμού. Βασίζεται και αυτό στην ύπαρξη μίας συλλογής από διεργασίες που αναλαμβάνουν τους υπολογισμούς. Οι τεχνικές σχεδιασμού των διεργασιών για την παραλληλοποίηση του σειριακού κώδικα είναι ίδιες με αυτές που είδαμε στο Κεφάλαιο 4. Όμως, σε αντίθεση με το μοντέλο κοινού χώρου διευθύνσεων, απαιτούνται μόνο δύο δομές για την συνεργασία των διεργασιών: μία για την αποστολή και μία για την λήψη μηνυμάτων. Ακριβώς αυτή η λιτότητα είναι που οδηγεί σε μακροσκελή, δυσανάγνωστα και δύσκολα στην αποσφαλμάτωση προγράμματα. Παρ' όλα αυτά το μοντέλο μεταβίβασης μηνυμάτων είναι σχεδόν μονόδρομος για τον προγραμματισμό υπολογιστικών συστάδων αλλά και των περισσότερων υπερυπολογιστικών συστημάτων, μιας και βασίζονται σε οργάνωση κατανεμημένης μνήμης. Τα συστήματα αυτά θεωρητικά μπορούν να προγραμματιστούν και με το μοντέλο κοινόχρηστου χώρου διευθύνσεων, χρησιμοποιώντας γλώσσες προγραμματισμού ή βιβλιοθήκες που στηρίζονται στην λεγόμενη κατανεμημένη κοινόχρηστη μνήμη λογισμικού (software DSM, [KKLR00]). Σε πολλές εφαρμογές έχει αποδειχτεί, όμως, ότι οι επιδόσεις που παίρνουμε δεν πλησιάζουν αυτές που προσφέρει το MPI. Μία πιο πρόσφατη προγραμματιστική επιλογή είναι το λεγόμενο μοντέλο τμηματοποιημένου καθολικού χώρου διευθύνσεων (Partitioned Global Address Space, PGAS), το οποίο προσπαθεί να παντρέψει την (κατανεμημένη) κοινόχρηστη μνήμη με τη μεταβίβαση μηνυμάτων, κυρίως με χρήση μονόπλευρων επικοινωνιών. Μία ανασκόπηση του μοντέλου, καθώς και των γλωσσών / βιβλιοθηκών για το μοντέλο αυτό δίνεται στο [WMFC15].

Ο προγραμματισμός με μεταβίβαση μηνυμάτων μπορεί να βελτιωθεί πολύ (τόσο σε αναγνωσιμότητα όσο και σε επιδόσεις) με την ύπαρξη επιπρόσθετων ευκολιών, όπως είναι οι κλήσεις για τις λεγόμενες συλλογικές επικοινωνίες που περιλαμβάνουν την εκπομπή, τη διασκόρπιση, τη συλλογή, τις λειτουργίες υποβίβασης κ.α. Οι επικοινωνίες αυτές γίνονται με τη συμμετοχή πολλών διεργασιών και αποδεικνύονται πολύτιμες στην πράξη. Επιπλέον, βελτιστοποιήσεις μπορούν να υπάρξουν με χρήση μη εμποδιστικών επικοινωνιών. Είναι χαρακτηριστικό ότι το MPI, που έχει εκτοπίσει όλες τις άλλες γλώσσες και βιβλιοθήκες μεταβίβασης μηνυμάτων, παρέχει όλες αυτές τις δυνατότητες και ακόμα περισσότερες.

Η μεταβίβαση μηνυμάτων, όπως είναι φυσικό, περιλαμβάνεται σχεδόν σε όλα τα βιβλία που ασχολούνται με τον παράλληλο προγραμματισμό (για παράδειγμα, [Quin03, WiAl04, LiSn08, Pach11]). Αν κάποιος θέλει να εμβαθύνει στη χρήση του MPI το ζεύγος βιβλίων [GLS14] και [GHTL14] αποτελεί έναν πολύ καλό οδηγό, με πολλά παραδείγματα, για τα πιο απλά και τα πιο προχωρημένα χαρακτηριστικά του. Στην ελληνική βιβλιογραφία, οι Πάντζιου, Μάμαλης και Τομάρας έχουν συμπεριλάβει και τον προγραμματισμό με το MPI στο βιβλίο τους [PMT13], ενώ υπάρχει και το βιβλίο του Α. Μάργαρη, [Marg07], το οποίο ασχολείται αποκλειστικά με το MPI και είναι ιδιαίτερα πλήρες. Εδώ, αξίζει να σημειώσουμε

ότι και το διαδίκτυο έχει κατακλυστεί από κείμενα αναφοράς, από εισαγωγικά μαθήματα (tutorials) αλλά και από παραδείγματα εφαρμογών γραμμένων σε MPI. Σε συνδυασμό με τη βιβλιογραφία, αποτελεί σημαντικό βοήθημα για όποιον θέλει να ασχοληθεί σοβαρά με το αντικείμενο.

Η ιστορία του MPI ξεκινά κάπου το 1992, όπου μία ομάδα ερευνητών και προγραμματιστών αποφάσισαν να ξεκινήσουν τη δημιουργία ενός ανοιχτού προτύπου για τη μεταβίβαση μηνυμάτων, το οποίο θα έβαζε ένα τέρμα στις πολλές και ασύμβατες υλοποιήσεις που υπήρχαν μέχρι τότε. Η πρώτη έκδοση του προτύπου δημοσιεύτηκε το 1994 και μέχρι το 1995 είχε φτάσει στην έκδοση 1.2, που ήταν για πολλά χρόνια η βασική του έκδοση.[6] Η επιτυχία του MPI ήταν άμεση και καθολική. Όλοι, μα όλοι, έσπευσαν να το υιοθετήσουν και μέσα σε λίγο καιρό έγινε συνώνυμο της μεταβίβασης μηνυμάτων. Πλέον, όταν κάποιος μιλά για αυτό το μοντέλο προγραμματισμού συνήθως εννοεί το MPI και μόνο. Η επικράτηση του MPI δεν ήταν καθόλου τυχαία.

- Ήταν πολύ καλά σχεδιασμένο και γενικό, και συγκέντρωνε την εμπειρία που υπήρχε μέχρι τότε από τις διάφορες ερευνητικές και εμπορικές υλοποιήσεις του μοντέλου.
- Ήταν πλήρες, και όχι μόνο. Από την πρώτη του έκδοση προσέφερε τεράστια ποικιλία επικοινωνιών (π.χ. συλλογικές, ασύγχρονες, μη εμποδιστικές) και άλλων ευκολιών.
- Ήταν ανοιχτό, με συμμετοχή και δικαίωμα γνώμης από όσους συμμετείχαν, αλλά και ελεύθερο. Κάθε έκδοση του προτύπου συνοδευόταν από έναν πλήρη οδηγό αναφοράς στις κλήσεις και τις λειτουργίες του, διαθέσιμο στον οποιονδήποτε.
- Συνοδεύτηκε από μία πολύ καλή υλοποίηση, αυτή του MPICH, η οποία ήταν επίσης ανοιχτού κώδικα. Ο κάθε κατασκευαστής μπορούσε να την χρησιμοποιήσει και να την φέρει στα μέτρα του δικού του συστήματος—κάτι που έγινε ευρέως.

Το 1997 οριστικοποιήθηκε η δεύτερη έκδοση του MPI, η οποία αποτελεί υπερσύνολο του MPI-1. Πρόσθεσε κυρίως δυναμική διαχείριση διεργασιών, MPI-IO και μονόπλευρες επικοινωνίες. Είναι χαρακτηριστικό, όμως, ότι μέχρι περίπου το 2005, που άρχισαν να εμφανίζονται πλήρεις και ελεύθερες υλοποιήσεις του, οι επιπλέον δυνατότητες που παρείχε δεν έτυχαν ευρείας αποδοχής. Ακόμα και σήμερα, η συμβατότητα με το MPI-1 είναι βασικό μέλημα στην ανάπτυξη εφαρμογών. Η τελευταία έκδοση του «MPI-2» είναι η 2.2 [MPI22] και εκδόθηκε το 2009.

Το MPI βρίσκεται πλέον στην τρίτη του έκδοση, από το 2012, ενώ τον Ιούνιο του 2015 δημοσιεύτηκε η έκδοση 3.1 [MPI31]. Το «MPI-3» πρόσθεσε, μεταξύ άλλων, νέες μονόπλευρες

[6] Η έκδοση 1.3 [MPI13], η οποία απλά ξεκαθαρίζει κάποιες λεπτομέρειες, είναι η τελευταία της σειράς 1.x, είναι αρκετά πιο πρόσφατη (2008), και θεωρείται πλέον ως το επίσημο πρότυπο «MPI-1».

λειτουργίες και μη εμποδιστικές εκδόσεις των συλλογικών επικοινωνιών.

Προβλήματα

5.1 – Δώστε ένα πρόγραμμα μεταβίβασης μηνυμάτων για τον υπολογισμό του π, το οποίο να χρησιμοποιεί τμηματική δρομολόγηση.

5.2 – Υποθέστε ότι διαθέτουμε N διεργασίες, κάθε μία από τις οποίες διαθέτει μία διαφορετική γραμμή ενός πίνακα `A[N][N]`, δηλαδή η κάθε διεργασία i διαθέτει την i-οστή γραμμή του πίνακα A. Ο αλγόριθμός μας απαιτεί σε κάποιο σημείο την *αναστροφή* του A, ούτως ώστε η διεργασία i να αναλάβει τελικά την i-οστή *στήλη*. Μπορείτε να σκεφτείτε τι επικοινωνίες απαιτούνται για να γίνει κάτι τέτοιο; Μπορούν να υλοποιηθούν με κάποια (ή κάποιες) συλλογικές επικοινωνίες;

5.3 – Έστω ο παρακάτω κώδικας, όπου υποθέτουμε ότι υπάρχουν 2 διεργασίες συνολικά, με ταυτότητες 0 και 1:

```
1 if (myid == 0) {
2   MPI_Recv(&b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
3   MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
4 }
5 else {
6   MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
7   MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
8 }
```

(α) Βλέπετε κάποιο πρόβλημα στον παραπάνω κώδικα;

(β) Αν χρησιμοποιούσαμε μη εμποδιστικές συναρτήσεις λήψης (`MPI_Irecv()`), θα υπήρχε καμία διαφορά;

5.4 – Προσπαθήστε να σκεφτείτε αν υπάρχει κάποιο πρόβλημα στις παρακάτω δύο εκδοχές του κώδικα του Προβλήματος 5.3:

(α) Η διεργασία 0 πρώτα κάνει την αποστολή και μετά τη λήψη (δηλαδή εναλλάσσονται οι γραμμές 2 και 3).

(β) Και η διεργασία 0 και η διεργασία 1 κάνουν πρώτα την αποστολή και μετά τη λήψη τους (δηλαδή εναλλάσσονται τόσοι οι γραμμές 2 και 3 όσο και οι 6, 7).

5.5 – Σε συνέχεια των Προβλημάτων 5.3 και 5.4, θεωρήστε ότι διαθέτουμε N διεργασίες οι οποίες εκτελούν ένα σενάριο παρόμοιο με αυτό που είδαμε στην Ενότητας 5.5 και το Σχ. 5.10. Συγκεκριμένα, οι N διεργασίες ανταλλάσσουν *κυκλικά* από ένα δεδομένο. Κάθε διεργασία i λαμβάνει ένα δεδομένο από την $i-1$ και στέλνει ένα δεδομένο στην $i+1$, με τη διεγρασία $N-1$ να στέλνει στην 0, και τη 0 να λαμβάνει από την $N-1$. Με ποια σειρά πρέπει η κάθε διεργασία να κάνει την αποστολή και τη λήψη της, ώστε να μην υπάρξει ποτέ πρόβλημα;
—*Σημείωση:* Μετά το πρόβλημα αυτό, ερευνήστε στη βιβλιογραφία (και το διαδίκτυο) για τη συνάρτηση `MPI_Sendrecv()` και τις χρήσεις της.

5.6 – Ανατρέχοντας στη συζήτηση της Ενότητας 5.5, μπορείτε να βρείτε τι πρόβλημα μπορεί να υπάρξει με τον κώδικα στο Σχ. 5.10; Μία βοήθεια υπάρχει στην υποσημείωση[7] αυτής της σελίδας. Στη συνέχεια, διορθώστε το πρόγραμμα, ώστε να λειτουργεί σίγουρα σωστά. Στους συνοδευτικούς κώδικες του βιβλίου περιέχεται η σωστή, ολοκληρωμένη εκδοχή, με την οποία μπορείτε να ελέγξετε τη λύση σας.

[7]Προσέξτε από πού φεύγει το μήνυμα με την `MPI_Isend()`. Είναι πάντοτε ασφαλής εκεί ο χώρος;

6 Μετρικές και Επιδόσεις

Στο κεφάλαιο αυτό θα δούμε με ποιον τρόπο προσδιορίζουμε τις επιδόσεις ενός παράλληλου αλγορίθμου / προγράμματος. Βασικός σκοπός μας επίσης είναι να δούμε πώς διαμορφώνονται αυτές οι επιδόσεις σε σχέση με μία σειριακή εκτέλεση. Έτσι, θα γίνουν φανερά τα πλεονεκτήματα (π.χ. επιτάχυνση στην εκτέλεση) αλλά και τα αναγκαία μειονεκτήματα (π.χ. καθυστερήσεις λόγω επικοινωνιών) που κάνουν την εμφάνισή τους μόνο στην παράλληλη λειτουργία. Τέλος, θα δούμε με ποιον τρόπο και υπό ποιους περιορισμούς μπορούμε να ανεβάσουμε τις επιδόσεις των προγραμμάτων μας χρησιμοποιώντας λιγότερους ή περισσότερους επεξεργαστές.

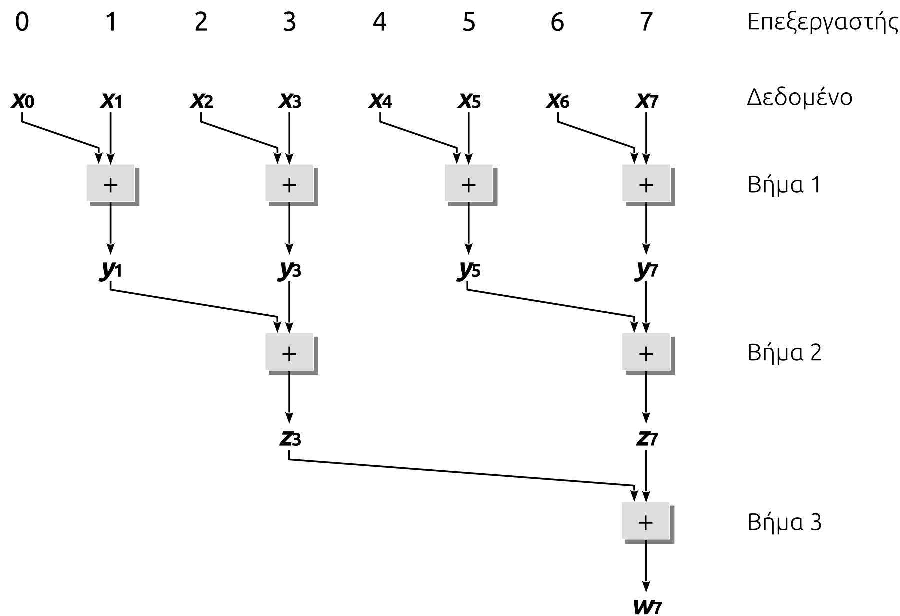
Ο λόγος ύπαρξης των παράλληλων υπολογιστών είναι η επιτάχυνση στη λύση προβλημάτων, πέρα από αυτό που μπορεί να δώσει ένας σειριακός υπολογιστής. Προκειμένου να κρίνουμε το μέγεθος αυτής της επιτάχυνσης και το κόστος με το οποίο θα επιτευχθεί, απαιτούνται κάποιες ποσότητες ή μετρικές. Ένας σειριακός αλγόριθμος γενικά «μετριέται» με βάση τον χρόνο εκτέλεσής του, που συνήθως είναι συνάρτηση του μεγέθους εισόδων του (π.χ. η σειριακή αναζήτηση απαιτεί χρόνο τάξης $O(N)$, αν η είσοδος έχει N μη ταξινομημένα στοιχεία). Ένας παράλληλος αλγόριθμος, όμως, εξαρτάται όχι μόνο από το μέγεθος της εισόδου του αλλά και από τον αριθμό των επεξεργαστών που τον εκτελούν. Στις Ενότητες 6.1–6.2 θα μελετήσουμε διάφορες μετρικές που προσδιορίζουν την επίδοση ενός παράλληλου αλγορίθμου. Θα δούμε, όμως, στην Ενότητα 6.3 πώς διαμορφώνονται αυτές οι μετρικές αν ληφθεί υπόψη και η επικοινωνιακή αρχιτεκτονική του παράλληλου συστήματος, η οποία αποτελεί βασικό στοιχείο επιβράδυνσης. Αυτό ακριβώς το αναγκαίο κακό προσπαθεί να μετριάσει ο χονδρόκοκκος παραλληλισμός με τη χρήση λιγότερων επεξεργαστών και άρα, λιγότερων επικοινωνιών. Όπως θα φανεί στην Ενότητα 6.4, ο μικρότερος αριθμός επεξεργαστών έχει φυσικά ως αποτέλεσμα μικρότερη ταχύτητα εκτέλεσης αλλά ταυτόχρονα μειώνει και τον άχρηστο χρόνο των επικοινωνιών, προσφέροντας έτσι μια πιο αποδοτική χρήση των επεξεργαστών που συμμετέχουν.

Ένας σειριακός αλγόριθμος που τροποποιείται για παράλληλη εκτέλεση δεν αποφέρει πάντα τα αναμενόμενα, κάτι που διατυπώνεται και ποσοτικά με τον νόμο του Amdahl, που αποτελεί το αντικείμενο της Ενότητας 6.5. Παρ' όλα αυτά, από το σύστημά μας μπορούμε να αντλήσουμε τις μέγιστες επιδόσεις που ζητάμε, αρκεί να το χρησιμοποιήσουμε για τη λύση του κατάλληλου (κλιμακωμένου όπως λέγεται) προβλήματος. Στην Ενότητα 6.6 θα δούμε αναλυτικά τον τρόπο με τον οποίο μπορεί να γίνει αυτό με βάση τον νόμο του Gustafson. Την έννοια της κλιμάκωσης θα την δούμε πιο αναλυτικά στην Ενότητα 6.8, αφού πρώτα προσδιορίσουμε στην Ενότητα 6.7 τους παράγοντες που ευθύνονται για τις αποκλίσεις που εμφανίζονται από τον επιθυμητό χρόνο εκτέλεσης ενός παράλληλου προγράμματος.

6.1 Χρόνος εκτέλεσης και επιτάχυνση

Ας υποθέσουμε ότι μελετάμε έναν αλγόριθμο με μέγεθος εισόδου N και ότι σε έναν σειριακό υπολογιστή αυτός απαιτεί χρόνο εκτέλεσης T_1. Ο χρόνος εκτέλεσης ενός παράλληλου αλγορίθμου προσδιορίζεται *από τη στιγμή που αρχίζουν οι υπολογισμοί μέχρι τη στιγμή που και ο τελευταίος επεξεργαστής ολοκληρώσει την εργασία του* και θα τον συμβολίσουμε με T_n, όπου n είναι ο αριθμός των επεξεργαστών.

Για να εκτιμήσουμε τον παράλληλο αλγόριθμο συνήθως μας ενδιαφέρει το όφελος που προκύπτει από την παραλληλοποίηση σε σχέση με μια σειριακή υλοποίηση. Επομένως,



Σχήμα 6.1 *Εύρεση του αθροίσματος 8 αριθμών*

μιλάμε για ένα συγκριτικό μέτρο το οποίο μας δίνει γενικά το κέρδος από την παράλληλη εκτέλεση. Η *επιτάχυνση* (speedup), S, ορίζεται από τον λόγο του σειριακού χρόνου προς τον παράλληλο:

$$S_n = \frac{T_1}{T_n}. \tag{6.1}$$

Για ένα δεδομένο πρόβλημα μπορεί να υπάρχουν πολλοί σειριακοί αλγόριθμοι που το λύνουν, οι οποίοι όμως να μην είναι κατάλληλοι για παραλληλοποίηση. Επομένως, υπάρχει κάποιο πρόβλημα στη σύγκριση με έναν παράλληλο αλγόριθμο: μπορεί να υπάρχει ένας πολύ αργός σειριακός αλγόριθμος ο οποίος όμως να επιδέχεται παραλληλοποίηση και άρα επιτάχυνση, όμως ακόμα και ο παράλληλος αυτός αλγόριθμος να εκτελείται πιο αργά από κάποιον άλλο καλύτερο σειριακό αλγόριθμο! Για τον λόγο αυτόν, είναι δίκαιο να συγκρίνουμε τον παράλληλο αλγόριθμο που εξετάζουμε με τον καλύτερο υπάρχοντα σειριακό αλγόριθμο για το δεδομένο πρόβλημα. Επομένως, στον ορισμό της επιτάχυνσης, ο όρος T_1 θα υποδηλώνει τον χρόνο που απαιτείται *για τον καλύτερο γνωστό σειριακό αλγόριθμο που λύνει το πρόβλημα*.

Παράδειγμα 6.1

» *Άθροισμα* N *αριθμών* $\{x_i \mid i = 0, 1, \ldots, N-1\}$ *με* $n = N$ *επεξεργαστές*

Θα υποθέσουμε για απλότητα ότι το n είναι δύναμη του 2, δηλαδή $n = 2^r$ για κάποιο r και άρα $r = \log n$, όπου ο λογάριθμος είναι με βάση το 2. Έστω λοιπόν, ότι έχουμε n επεξεργαστές και ότι ο επεξεργαστής i χειρίζεται τον αριθμό x_i. Για να βρούμε το άθροισμα εκτελούμε τον εξής απλό αλγόριθμο, που φαίνεται στο Σχ. 6.1 για $n = 8$

στοιχεία.

Στο πρώτο βήμα οι επεξεργαστές 0, 2, 4, ... στέλνουν τον αριθμό τους στους επεξεργαστές 1, 3, 5, ...αντίστοιχα, οι οποίοι υπολογίζουν το άθροισμα αυτού που διαθέτουν και αυτού που λαμβάνουν. Έτσι, στο τέλος του πρώτου βήματος έχουμε το ίδιο πρόβλημα μόνο που τώρα έχουν μείνει $n/2$ αριθμοί y_1, y_3, y_5, ...στους επεξεργαστές 1, 3, 5, ...Παρόμοια, στο δεύτερο βήμα οι επεξεργαστές 1, 5, 9, ... στέλνουν το αποτέλεσμα αντίστοιχα στους 3, 7, 11, ... όπου υπολογίζεται πάλι το άθροισμα αυτού που διαθέτουν και αυτού που λαμβάνουν. Προκύπτουν έτσι $n/4$ αριθμοί z_3, z_7, z_{11}, ... στους αντίστοιχους επεξεργαστές. Γενικά,

> κατά το βήμα j κάθε επεξεργαστής i, όπου $i = \{1 \times 2^{j-1} - 1, 3 \times 2^{j-1} - 1, 5 \times 2^{j-1} - 1, \ldots\}$, στέλνει τον αριθμό που διαθέτει στον επεξεργαστή $i + 2^{j-1}$.

Είναι εύκολο να δούμε ότι σε κάθε βήμα το πλήθος των αριθμών που απομένουν μειώνεται στο μισό και το άθροισμα θα βρεθεί τελικά στον επεξεργαστή $n-1$ μετά από $r = \log n$ βήματα. Υποθέτοντας ότι για να αποσταλούν τα δεδομένα από έναν επεξεργαστή σε έναν άλλο απαιτείται σταθερός χρόνος (πολύ σημαντική και όχι πάντα σωστή υπόθεση όπως θα δούμε αργότερα), ο συνολικός χρόνος για τον παράλληλο αλγόριθμό μας θα είναι:

$$T_n = O(\log n).$$

Από την άλλη μεριά, γνωρίζουμε ότι ένας σειριακός αλγόριθμος για το ίδιο πρόβλημα απαιτεί $T_1 = O(n)$ βήματα. Επομένως,

$$S_n = O\left(\frac{n}{\log n}\right).$$

6.1.1 Γραμμική και υπεργραμμική επιτάχυνση

Θεωρητικά, πάντα θα πρέπει να ισχύει ότι $S_n \le n$, δηλαδή η επιτάχυνση είναι το πολύ *γραμμική*, και αυτό γιατί δεν είναι δυνατόν ο χρόνος παράλληλης εκτέλεσης να είναι μικρότερος του T_1/n. Αν ήταν μικρότερος, τότε θα μπορούσαμε να εκτελούσαμε σειριακά την εργασία που κάνουν οι n επεξεργαστές και να συνθέσουμε έτσι ένα σειριακό πρόγραμμα με χρόνο εκτέλεσης μικρότερο του T_1. Μερικές φορές όμως, στην πράξη, μπορεί να παρατηρηθεί το φαινόμενο της *υπεργραμμικής επιτάχυνσης* (superlinear speedup), όπου $S_n > n$. Οι λόγοι που μπορεί να οδηγήσουν σε αυτό το παράδοξο, είναι:

(α) Ο καλύτερος σειριακός αλγόριθμος δεν είναι απόλυτα βέλτιστος. Αυτό μπορεί να έχει ως αποτέλεσμα η παραλληλοποίησή του ή η παραλληλοποίηση ενός άλλου (κατώτερου) σειριακού αλγορίθμου να επιφέρει καλύτερη συμπεριφορά στα υποτμήματα των υπολογισμών ή στις δομές δεδομένων που χρησιμοποιούνται και το παράλληλο πρόγραμμα θα ξεπερνά το n σε επιτάχυνση

(β) Η τυχαιότητα του αλγορίθμου. Υπάρχουν αλγόριθμοι που αναζητούν τη λύση του προβλήματος κάνοντας τυχαία αναζήτηση ανάμεσα σε ένα σύνολο πιθανών λύσεων. Η παράλληλη αναζήτηση, μπορεί τυχαία να εντοπίσει τη λύση με ελάχιστα βήματα, σε συγκεκριμένες περιπτώσεις.

(γ) Το μέγεθος του συστήματος επηρεάζει την ταχύτητα. Ένα κλασικό παράδειγμα είναι οι πολύ μεγάλες εφαρμογές με δεδομένα που δεν χωρούν στην κύρια μνήμη ενός σειριακού υπολογιστή, με αποτέλεσμα να γίνονται προσπελάσεις και στο δίσκο. Όμως, σε έναν παράλληλο υπολογιστή με επεξεργαστές που διαθέτουν ιδιωτικές μνήμες, τα δεδομένα όταν διασπαστούν ίσως χωρούν στις ιδιωτικές μνήμες και άρα, θα αποφευχθούν οι προσπελάσεις στον δίσκο, βελτιώνοντας έτσι την ταχύτητα.

Το φαινόμενο αυτό είναι καλό να το γνωρίζουμε, όμως δεν πρόκειται να ασχοληθούμε μαζί του παραπέρα.

6.2 Αποδοτικότητα και κόστος

Η επιτάχυνση μετρά το πόσο γρηγορότερα μπορεί να εκτελεστεί ένα πρόγραμμα σε έναν παράλληλο υπολογιστή απ' ό,τι σε έναν σειριακό. Όμως, δεν μας λέει τίποτε για το πόσο χρήσιμοι είναι οι επεξεργαστές στη βελτίωση αυτή. Θα ήταν αρκετά απογοητευτικό, αν επρόκειτο να χρειαστούν 1000 επεξεργαστές για να εκτελέσουμε ένα πρόγραμμα 2-3 φορές γρηγορότερα. Η *αποδοτικότητα* (efficiency) εξυπηρετεί αυτόν ακριβώς το σκοπό· σχετίζει το όφελος στην ταχύτητα με το πλήθος των επεξεργαστών που χρειάζονται για να το επιτύχουμε:

$$e_n = \frac{S_n}{n} = \frac{T_1}{nT_n}. \tag{6.2}$$

Προσέξτε ότι από τη στιγμή που $S_n \leq n$, θα έχουμε

$$e_n \leq 1,$$

δηλαδή το πολύ να υπάρχει αποδοτικότητα 100% (ιδανική), όπου όλοι οι επεξεργαστές όντως είναι απαραίτητοι προκειμένου να επιτευχθεί η επιτάχυνση S_n. Στο Παράδειγμα 6.1

είμαστε βέβαιοι ότι η αποδοτικότητα θα είναι κάτω από το 100%, αφού σε πολλά βήματα υπάρχουν επεξεργαστές οι οποίοι δεν κάνουν τίποτε. Πράγματι, ισχύει:

$$e_n = O\left(\frac{1}{\log n}\right) < 1.$$

Είναι λογικό να επιθυμούμε προγράμματα για τα οποία ισχύει $S_n = n$ ή ισοδύναμα $e_n = 1$ (100%). Τέτοια προγράμματα ονομάζονται *πλήρως παραλληλοποιήσιμα*.

Ένα άλλο μέγεθος που μας δείχνει κατά πόσο οι επεξεργαστές χρησιμοποιούνται αποδοτικά είναι το *κόστος* του προγράμματος, το οποίο ορίζεται ως το γινόμενο του παράλληλου χρόνου εκτέλεσης και του αριθμού των επεξεργαστών:

$$c_n = nT_n. \tag{6.3}$$

Με άλλα λόγια, το κόστος μας δείχνει πόση ώρα χρησιμοποιήθηκαν οι επεξεργαστές αθροιστικά. Αν ο αλγόριθμος εκτελέστηκε σε χρόνο T_n, τότε κάθε επεξεργαστής ήταν δεσμευμένος για χρόνο T_n (άσχετα αν δεν εργαζόταν σε όλο αυτό το διάστημα) και άρα, αθροιστικά ο χρόνος που αφιερώθηκε στον αλγόριθμο από τους n επεξεργαστές ήταν nT_n. Το κόστος της σειριακής εκτέλεσης είναι προφανώς ίσο με T_1. Αν όλοι οι επεξεργαστές εργάζονταν όλη την ώρα και το πρόγραμμα ήταν πλήρως παραλληλοποιήσιμο (δηλαδή η επιτάχυνση ήταν ίση με n και άρα $T_n = T_1/n$), τότε το κόστος της παράλληλης εκτέλεσης θα ήταν ίσο με αυτό της σειριακής. Επειδή όμως η επιτάχυνση μπορεί να είναι μικρότερη του n, το κόστος της παράλληλης εκτέλεσης είναι συνήθως μεγαλύτερο από αυτό του σειριακού. Στο Παράδειγμα 6.1, θα έχουμε:

$$c_n = nO(\log n) = O(n \log n),$$

το οποίο είναι μεγαλύτερο από το $T_1 = O(n)$. Ένα πρόγραμμα θα έχει *βέλτιστο κόστος* όταν το κόστος του, διαιρούμενο με το σειριακό κόστος, είναι ένας σταθερός αριθμός. Συμβολικά, βέλτιστο κόστος επιτυγχάνεται όταν:

$$\frac{c_n}{T_1} = O(1).$$

6.3 Εξάρτηση από την αρχιτεκτονική

Στην ενότητα αυτή, θα δούμε τη σημασία που έχει η αρχιτεκτονική οργάνωση του συστήματος και κυρίως το επικοινωνιακό υποσύστημα, στον χρόνο εκτέλεσης ενός παράλληλου προγράμματος.

Ως επί το πλείστον, κατά τη διάρκεια εκτέλεσης των παράλληλων προγραμμάτων θα πρέπει να υπάρχει επικοινωνία ή και συντονισμός των επεξεργαστών. Σε ένα μηχάνημα κοινής μνήμης η επικοινωνία είναι έμμεση, μέσω κοινών μεταβλητών. Η προσπέλαση

όμως των κοινών μεταβλητών εισάγει καθυστερήσεις, δηλαδή χρόνο ο οποίος σπαταλάται σε ενέργειες που δεν έχουν σχέση με τους καθαυτό υπολογισμούς του αλγορίθμου. Όπως είδαμε στο Κεφάλαιο 4, η προσπέλαση κοινών μεταβλητών απαιτεί μηχανισμούς αμοιβαίου αποκλεισμού οι οποίοι ουσιαστικά σειριοποιούν τις ταυτόχρονες προσπελάσεις από διαφορετικούς επεξεργαστές. Επίσης, σε διάφορα σημεία του προγράμματος απαιτείται συγχρονισμός των επεξεργαστών. Έτσι, στο Παράδειγμα 6.1, οι επεξεργαστές θα πρέπει όλοι να εκτελούν το ίδιο βήμα για να έχει νόημα το τελικό αποτέλεσμα. Για παράδειγμα, ο επεξεργαστής 7 θα πρέπει να περιμένει τον επεξεργαστή 3 να ολοκληρώσει το δεύτερο βήμα πριν προχωρήσει στο τρίτο βήμα. Ο συγχρονισμός επιτυγχάνεται σε κατάλληλα σημεία της εκτέλεσης μέσω κάποιας συνεννόησης των επεξεργαστών, πράγμα το οποίο εισάγει επιπλέον καθυστερήσεις.

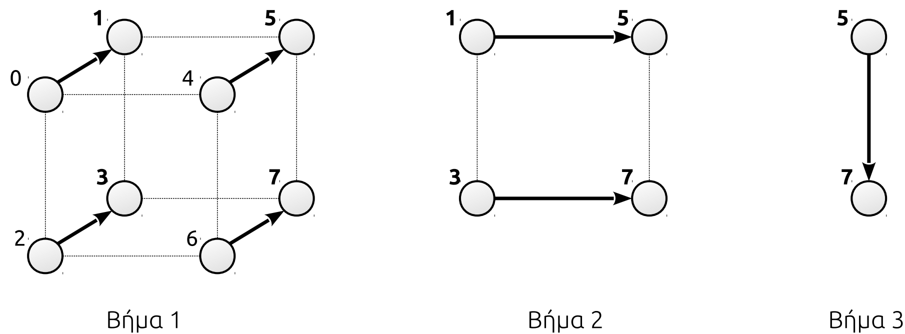
Σε έναν πολυεπεξεργαστή κατανεμημένης μνήμης οι καθυστερήσεις λόγω της αρχιτεκτονικής διάταξης του συστήματος είναι πολύ πιο ξεκάθαρες. Προκειμένου να επέλθει συνεννόηση μεταξύ των επεξεργαστών απαιτείται η αποστολή και η λήψη μηνυμάτων, ενέργειες που χρειάζονται μη αμελητέο χρόνο για να πραγματοποιηθούν. Το δίκτυο διασύνδεσης παίζει τον σημαντικότερο ρόλο στην καθυστέρηση αυτή. Έχουν χρησιμοποιηθεί αρκετές τεχνικές για την μείωση των καθυστερήσεων π.χ. μέσω βελτίωσης στη διάμετρο του δικτύου διασύνδεσης ή μέσω επιτάχυνσης του τρόπου μετάδοσης των μηνυμάτων (βελτιωμένη μεταγωγή). Παρά τις όποιες βελτιώσεις, το ζητούμενο είναι να αποφεύγεται η χρήση του δικτύου όσο το δυνατόν περισσότερο, καθώς ακόμα και στην καλύτερη περίπτωση η μετάδοση ενός μηνύματος απαιτεί χρόνο τάξης μεγέθους μεγαλύτερο από αυτόν που απαιτείται για έναν απλό υπολογισμό (π.χ. μία πρόσθεση).

Παράδειγμα 6.2

» *Πρόσθεση N αριθμών σε γραμμικό γράφο με $n = N$ επεξεργαστές*

Θα ακολουθήσουμε τον αλγόριθμο στο Παράδειγμα 6.1. Θα υποθέσουμε επίσης ότι ο χρόνος που απαιτείται για μία πράξη (πρόσθεση) είναι 1 χρονική μονάδα και ότι ο χρόνος για να μεταφερθεί ένας αριθμός από έναν επεξεργαστή σε έναν γείτονά του (με τον οποίο είναι άμεσα συνδεδεμένος) είναι επίσης 1 χρονική μονάδα.

Κατά το βήμα 1, ο επεξεργαστής i επικοινωνεί με τον επεξεργαστή $i + 1$. Όμως, στον γραμμικό γράφο οι επεξεργαστές αυτοί είναι γειτονικοί. Άρα, χρειάζεται μόνο 1 χρονική μονάδα για να μεταφέρει ο επεξεργαστής i τον αριθμό x_i στον επεξεργαστή $i+1$. Στο δεύτερο βήμα, ένας επεξεργαστής i που συμμετέχει πρέπει να στείλει τον αριθμό του στον επεξεργαστή $i+2$ ο οποίος βρίσκεται σε απόσταση 2. Γενικά, όπως είδαμε, στο βήμα j αν ο επεξεργαστής i συμμετέχει πρέπει να στείλει τον αριθμό του στον επεξεργαστή $i+2^{j-1}$, ο οποίος βρίσκεται σε απόσταση 2^{j-1}. Άρα, στο βήμα j οι επικοινωνίες απαιτούν 2^{j-1} χρονικές μονάδες.



Σχήμα 6.2 *Επικοινωνίες για πρόσθεση 8 αριθμών σε τρισδιάστατο υπερκύβο*

Συνολικά για όλο τον αλγόριθμο ($\log n$ βήματα) ο χρόνος που θα απαιτηθεί είναι:

$$T_n = \sum_{j=1}^{\log n} (1 + 2^{j-1}),$$

όπου 1 χρονική μονάδα απαιτείται για την πρόσθεση και 2^{j-1} χρονικές μονάδες για την επικοινωνία. Το άθροισμα αυτό, αν υπολογιστεί, δίνει:

$$T_n = n + \log n - 1,$$

όπου με έκπληξη διαπιστώνεται ότι στον γραμμικό γράφο η επικοινωνία των επεξεργαστών είναι τόσο χρονοβόρα, ώστε ο αλγόριθμος είναι πιο αργός από τον σειριακό!

Παράδειγμα 6.3

» *Πρόσθεση N αριθμών σε υπερκύβο με $n = N = 2^d$ επεξεργαστές*

Στον υπερκύβο τα πράγματα είναι πολύ διαφορετικά. Έστω ότι ο επεξεργαστής i χειρίζεται αρχικά τον αριθμό x_i. Στο βήμα j, όπως είδαμε στο Παράδειγμα 6.1, κάθε επεξεργαστής i (όπου $i = 1 \times 2^{j-1} - 1, 3 \times 2^{j-1} - 1, 5 \times 2^{j-1} - 1$, ...θα στείλει τον αριθμό που διαθέτει στον επεξεργαστή $k = i+2^{j-1}$. Μπορεί να δει εύκολα κανείς ότι η δυαδική αναπαράσταση του i περιέχει 0 στο j-οστό bit. Ο αριθμός 2^{j-1} περιέχει στη δυαδική του αναπαράσταση μόνο μηδενικά, εκτός από το j-οστό bit που είναι ίσο με 1. Άρα, η δυαδική αναπαράσταση του k είναι ίδια με του i, με τη διαφορά ότι το j-οστό bit έχει πάρει την τιμή 1. Επομένως, οι επεξεργαστές i και k είναι γειτονικοί στον υπερκύβο, αφού διαφέρουν μόνο σε ένα bit (δείτε και το Σχ. 6.2). Το συμπέρασμα είναι ότι η επικοινωνία σε κάθε βήμα j απαιτεί *μόνο μία χρονική μονάδα*. Επομένως, συνολικά ο αλγόριθμος θα απαιτήσει χρόνο ίσο με

$$T_n = 2 \log n,$$

αφού σε κάθε βήμα χρειάζεται μία χρονική μονάδα για προσθέσεις και μία χρονική μονάδα για επικοινωνία.

Το Παράδειγμα 6.2 και το Παράδειγμα 6.3 δείχνουν πόσο μεγάλο ρόλο παίζουν οι επικοινωνίες στον χρόνο εκτέλεσης. Θα πρέπει, όμως, να προσθέσουμε κάτι που δεν φαίνεται άμεσα από τα παραπάνω. Η αντιστοίχιση των διαφόρων εργασιών στους επεξεργαστές, που περιγράψαμε στο Κεφάλαιο 1, παίζει επίσης σημαντικό ρόλο. Εάν στο Παράδειγμα 6.3 ο αριθμός x_2 φυλασσόταν στον επεξεργαστή 4 (όχι δηλαδή στον 2) και ο αριθμός x_3 φυλασσόταν στον επεξεργαστή 3, τότε στο πρώτο βήμα θα χρειάζονταν τρεις χρονικές μονάδες για τη μεταφορά του x_2, αφού οι επεξεργαστές 4 και 3 έχουν απόσταση τρία στον υπερκύβο.

Ως γενικό συμπέρασμα, επομένως, θα πρέπει να ειπωθεί ότι η αρχιτεκτονική του συστήματος και ιδιαίτερα το επικοινωνιακό σύστημα είναι βασικός παράγοντας καθυστερήσεων. Γι' αυτό και προτιμούνται συστήματα με ισχυρά διασυνδετικά δίκτυα (π.χ. υπερκύβος). Όμως, η πλήρης εκμετάλλευση μιας δεδομένης αρχιτεκτονικής γίνεται μέσα από σωστές αντιστοιχίσεις των εργασιών στους επεξεργαστές.

6.4 Λίγοι επεξεργαστές—αύξηση κόκκου παραλληλίας

Μέχρι τώρα είδαμε παραδείγματα όπου ο αριθμός των επεξεργαστών, n, ήταν ίσος με το μέγεθος εισόδου του προβλήματος, N. Τι γίνεται όμως στην πιο ρεαλιστική περίπτωση όπου διαθέτουμε ένα σχετικά μικρό πλήθος επεξεργαστών για τη λύση ενός μεγάλου προβλήματος; Με αυτή την περίπτωση θα ασχοληθούμε στην τρέχουσα ενότητα. Θα δούμε τη βασική τεχνική των εικονικών επεξεργαστών και τον τρόπο με τον οποίο επηρεάζονται οι επιδόσεις του παράλληλου προγράμματος.

Οι εμπορικές παράλληλες μηχανές συνήθως διαθέτουν έναν σχετικά μικρό αριθμό επεξεργαστών. Ακόμα όμως και να είχαμε στη διάθεσή μας ένα ερευνητικό σύστημα με χιλιάδες επεξεργαστές, τα προβλήματα που επιθυμούμε να λύσουμε (δείτε για παράδειγμα το πρόβλημα των N σωμάτων στο Κεφάλαιο 1) ξεπερνούν κατά πολύ σε μέγεθος τον αριθμό των επεξεργαστών.

Ο αρχικός αλγόριθμος, τουλάχιστον σε θεωρητικό επίπεδο, σχεδιάζεται κατά κανόνα ανεξάρτητα από τη μηχανή που θα τον εκτελέσει. Γίνεται η υπόθεση ότι διαθέτουμε όσους επεξεργαστές θελήσουμε, δηλαδή ουσιαστικά εργαζόμαστε με ένα απεριόριστο πλήθος επεξεργαστών. Κάθε ένας από αυτούς τους επεξεργαστές αναλαμβάνει ένα (ή παραπάνω) στοιχείο της εισόδου του προβλήματος και ονομάζεται *εικονικός επεξεργαστής* (virtual processor), σε αντιδιαστολή με τους *πραγματικούς* επεξεργαστές που διαθέτει ο υπολογιστής ο οποίος θα φιλοξενήσει το πρόγραμμα. Έστω λοιπόν ότι το μέγεθος εισόδου και ο αριθμός των εικονικών επεξεργαστών είναι N, ενώ η μηχανή που διαθέτουμε έχει

$n < N$ επεξεργαστές.

Η βασική τεχνική υλοποίησης του προγράμματος είναι η *προσομοίωση* των εικονικών επεξεργαστών από τους πραγματικούς: κάθε πραγματικός επεξεργαστής αναλαμβάνει να εκτελέσει όλες τις ενέργειες (υπολογιστικές και επικοινωνιακές) N/n εικονικών επεξεργαστών. Θα πρέπει να είναι αυτονόητο ότι με αυτόν τον τρόπο αυξάνει ο χρόνος εκτέλεσης. Συγκεκριμένα, από τη στιγμή που ο κάθε επεξεργαστής εκτελεί σειριακά πλέον τους υπολογισμούς N/n εικονικών επεξεργαστών, ο χρόνος των καθαυτό υπολογισμών θα αυξηθεί κατά έναν παράγοντα N/n.

Θα αυξηθεί επίσης και ο χρόνος των επικοινωνιών ανά επεξεργαστή. Αν κάθε εικονικός επεξεργαστής σπαταλούσε κατά μέσο όρο χρόνο t για επικοινωνίες, σε έναν πραγματικό επεξεργαστή ο επικοινωνιακός χρόνος θα είναι επίσης αυξημένος κατά έναν παράγοντα N/n, εξομοιώνοντας τα βήματα των εικονικών επεξεργαστών. Υπάρχει βέβαια και η περίπτωση, ο χρόνος επικοινωνίας να είναι αυξημένος πιο πολύ από N/n φορές, αν οι εικονικοί επεξεργαστές έχουν ανατεθεί ανορθόδοξα στους πραγματικούς επεξεργαστές. Όμως στην πράξη, η διαμοίραση των εικονικών επεξεργαστών μπορεί να γίνει με τέτοιον τρόπο, ώστε ο χρόνος επικοινωνίας να μην αυξηθεί περισσότερο από N/n φορές και μάλιστα, όπως θα δούμε παρακάτω, η αύξηση να είναι ακόμα μικρότερη.

Γενικά λοιπόν, βλέπουμε ότι τόσο ο χρόνος υπολογισμών όσο και ο χρόνος επικοινωνιών για κάθε επεξεργαστή είναι το πολύ N/n φορές μεγαλύτερος για έναν πραγματικό επεξεργαστή σε σχέση με τους αντίστοιχους χρόνους ενός εικονικού επεξεργαστή. Επομένως, ο χρόνος εκτέλεσης του προγράμματος σε n επεξεργαστές θα είναι αυξημένος το πολύ κατά έναν παράγοντα N/n σε σχέση με τον χρόνο εκτέλεσης σε (εικονικούς) επεξεργαστές, δηλαδή ο αναμενόμενος χρόνος εκτέλεσης θα είναι:

$$T_n = O\left(\frac{N}{n}T_N\right).$$

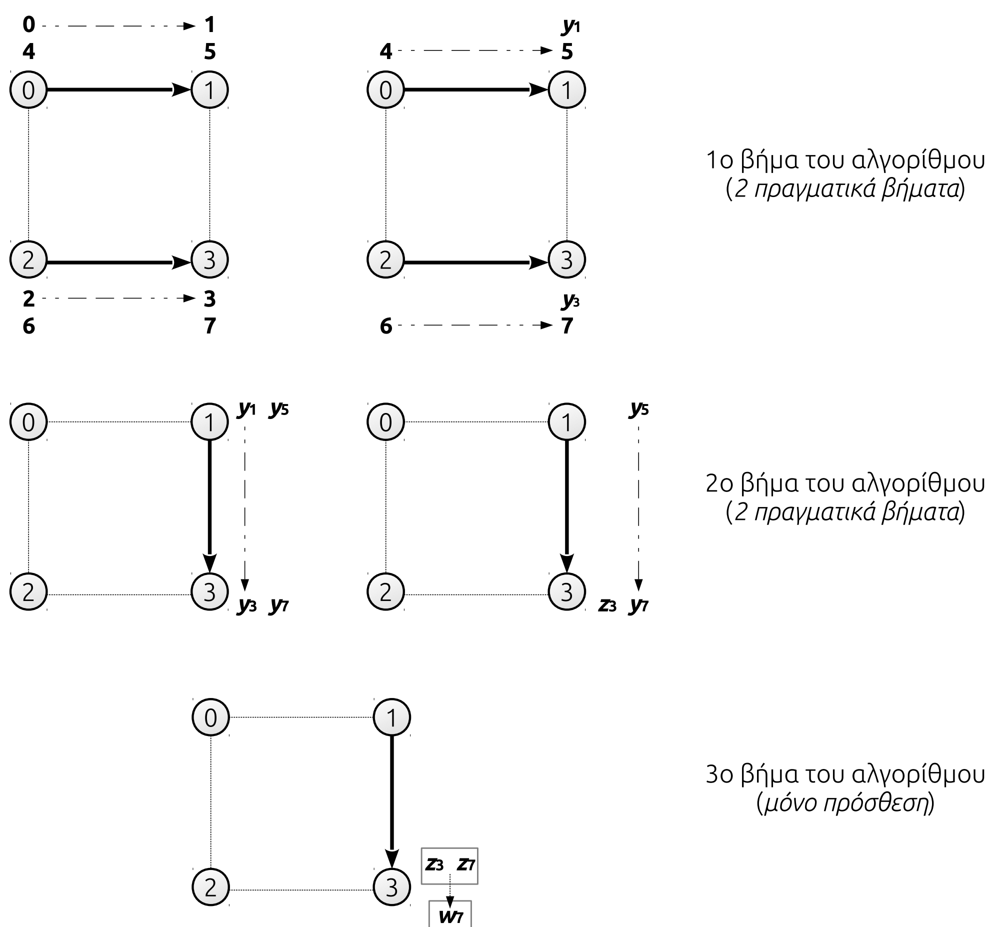
Κατ' επέκταση και χρησιμοποιώντας τους τύπους (6.1)–(6.3), δεν είναι δύσκολο να δούμε ότι η επιτάχυνση S_n θα είναι ίση με $O((n/N)S_N)$ και η αποδοτικότητα ίση με $e_n = O(e_N)$. Το ίδιο ισχύει και για το κόστος του προγράμματος, αφού:

$$c_n = nT_n = nO\left(\frac{N}{n}T_N\right) = O(NT_N) = O(c_N).$$

Όλες αυτές οι σχέσεις τονίζουν το εξής γεγονός: *η μείωση του αριθμού επεξεργαστών δεν μειώνει την αποδοτικότητα και δεν αυξάνει το κόστος του αλγορίθμου*. Έτσι, για παράδειγμα, ένας αλγόριθμος βέλτιστου κόστους θα έχει επίσης βέλτιστο κόστος, αν υλοποιηθεί με λιγότερους επεξεργαστές.

Παράδειγμα 6.4

» *Πρόσθεση N αριθμών σε υπερκύβο με $n < N$ επεξεργαστές*



Σχήμα 6.3 *Πρόσθεση 8 αριθμών σε διδιάστατο υπερκύβο (4 κόμβων)*

Υποθέτουμε πάλι για ευκολία, ότι οι αριθμοί N και n είναι δυνάμεις του 2, ώστε ο λόγος N/n να είναι ακέραιος. Θα χρησιμοποιήσουμε τον αλγόριθμο από το Παράδειγμα 6.1, όπως τον εφαρμόσαμε στον υπερκύβο στο Παράδειγμα 6.3 και θα αναθέσουμε τον εικονικό επεξεργαστή i στον επεξεργαστή $(i \bmod n)$ του υπερκύβου. Έτσι, κάθε επεξεργαστής αναλαμβάνει N/n εικονικούς επεξεργαστές. Για παράδειγμα, ο επεξεργαστής 0 αναλαμβάνει τους εικονικούς επεξεργαστές 0, n, $2n$, ..., ενώ ο επεξεργαστής 1 αναλαμβάνει τους εικονικούς επεξεργαστές 1, $n+1$, $2n+1$ κλπ. Στο Σχ. 6.3 έχουμε έναν διδιάστατο υπερκύβο για την πρόσθεση 8 αριθμών. Οι πραγματικοί επεξεργαστές είναι αριθμημένοι από 0 έως 3 και καθένας από αυτούς αναλαμβάνει δύο ιδεατούς επεξεργαστές που σημειώνονται με έντονα γράμματα.

Με αυτόν τον τρόπο ανάθεσης, μπορούμε να δούμε ότι, κατά τη διάρκεια των πρώτων $\log n$ βημάτων του αλγορίθμου, κάθε επεξεργαστής εκτελεί την εργασία των N/n εικονικών επεξεργαστών που έχει αναλάβει και μάλιστα ένα βήμα του αρχικού αλγορίθμου προσομοιώνεται από N/n βήματα στον υπερκύβο μας. Μετά το

πέρας αυτών των βημάτων, όλοι οι N/n αριθμοί που έχουν απομείνει βρίσκονται σε έναν μόνο επεξεργαστή. Ως αποτέλεσμα, από τη στιγμή αυτή και ύστερα δεν υπάρχουν επικοινωνίες και ο μοναδικός ενεργός επεξεργαστής χρειάζεται μόνο να κάνει $N/n - 1$ προσθέσεις. Δείτε το Σχ. 6.3, όπου φαίνονται όλα τα βήματα της πρόσθεσης 8 αριθμών σε έναν υπερκύβο με 4 επεξεργαστές.

Βλέπουμε λοιπόν ότι πλέον ο χρόνος εκτέλεσης διαμορφώνεται σε $O((N/n)\log n)$ για τα πρώτα $\log n$ βήματα του αρχικού αλγορίθμου και $O(N/n)$ για τα υπόλοιπα, δηλαδή συνολικά, $T_n = O((N/n)\log n)$. Το κόστος του νέου προγράμματος είναι $c_n = nT_n = O(N \log n)$, το οποίο δεν είναι βέλτιστο, όπως συνέβαινε και στον αρχικό αλγόριθμο.

Αυτό που κάνουμε όταν εκτελούμε τον αλγόριθμο με λιγότερους επεξεργαστές είναι στην ουσία η *αύξηση του κόκκου παραλληλίας* (θυμηθείτε αυτά που είπαμε στο Κεφάλαιο 1). Από τη στιγμή που ένας πραγματικός επεξεργαστής «εμπεριέχει» αρκετούς εικονικούς επεξεργαστές, εκτελεί περισσότερα υπολογιστικά βήματα. Ταυτόχρονα όμως, από τη στιγμή που ορισμένοι εικονικοί επεξεργαστές έχουν τοποθετηθεί στον ίδιο πραγματικό επεξεργαστή, υπάρχει περίπτωση να μην χρειάζονται ορισμένα επικοινωνιακά βήματα. Αυτό ακριβώς συνέβη και στο Παράδειγμα 6.4, όπου ο χρόνος εκτέλεσης διαμορφώθηκε ως $T_n = O((N/n)\underline{\log n})$ ενώ, από όσα είπαμε στην ενότητα αυτή, ο αναμενόμενος χρόνος εκτέλεσης ήταν $O((N/n)T_N)$, δηλαδή $O((N/n)\underline{\log N})$. Ο χρόνος εκτέλεσης τελικά ήταν μικρότερος από τον αναμενόμενο.

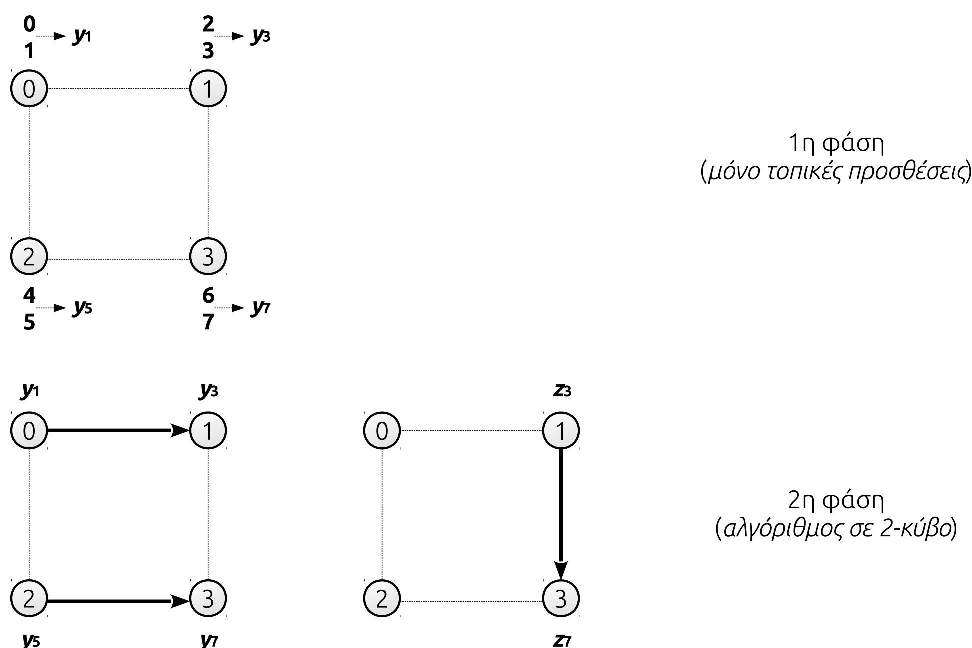
Βλέπουμε, λοιπόν, ότι ο χρόνος που αφιερώνεται για τις επικοινωνίες μπορεί να είναι αυξημένος κατά έναν παράγοντα μικρότερο του N/n, δηλαδή μπορεί να αυξάνει με μικρότερο ρυθμό απ' ό,τι αυξάνει ο χρόνος των καθαρών υπολογισμών ανά επεξεργαστή.

Η τελευταία παρατήρηση είναι ιδιαίτερα σημαντική και εξηγεί τα θετικά σημεία της χονδρόκοκκης παραλληλίας. Σε ορισμένες περιπτώσεις μάλιστα, ο μειωμένος ρυθμός αύξησης των επικοινωνιακών χρόνων είναι ικανός να δημιουργήσει ένα πρόγραμμα με βέλτιστο κόστος από έναν αρχικό μη βέλτιστο αλγόριθμο. Αυτό ακριβώς δείχνει το επόμενο παράδειγμα. Το κλειδί για μια τέτοια βελτίωση είναι η μείωση των αναγκαίων επικοινωνιών στο ελάχιστο, μέσα από μία μελετημένη *ανάθεση* των εικονικών επεξεργαστών στους πραγματικούς.

Παράδειγμα 6.5

» *Πρόσθεση N αριθμών σε υπερκύβο $n < N$ επεξεργαστών με βέλτιστο κόστος*

Υποθέτουμε, όπως στο Παράδειγμα 6.4, ότι οι αριθμοί N και n είναι δυνάμεις του 2, ώστε ο λόγος N/n να είναι ακέραιος, με τη διαφορά ότι τώρα θα κάνουμε διαφορετική ανάθεση των εικονικών επεξεργαστών. Συγκεκριμένα, ο πραγματι-



Σχήμα 6.4 *Νέα πρόσθεση 8 αριθμών σε διδιάστατο υπερκύβο*

κός επεξεργαστής i θα αναλάβει τους N/n συνεχόμενους εικονικούς επεξεργαστές $i \times (N/n), i \times (N/n)+1, i \times (N/n)+2, \ldots, i \times (N/n)+(N/n)-1$. Έτσι, ο επεξεργαστής 0 αναλαμβάνει τους εικονικούς επεξεργαστές 0, 1, 2, ..., $N/n-1$, ο επεξεργαστής 1 αναλαμβάνει τους επόμενους N/n εικονικούς επεξεργαστές κλπ. Στο Σχ. 6.4 φαίνεται η νέα ανάθεση για τον 2-κύβο.

Προσέξτε τώρα ότι στα πρώτα βήματα του αλγορίθμου όλες οι προσθέσεις είναι τοπικές, δηλαδή οι συμμετέχοντες εικονικοί επεξεργαστές βρίσκονται όλοι στον ίδιο επεξεργαστή. Επομένως, δεν απαιτούνται επικοινωνιακά βήματα. Ο κάθε πραγματικός επεξεργαστής απλά προσθέτει τους N/n αριθμούς που έχει αναλάβει, κάνοντας $N/n-1$ προσθέσεις, σε χρόνο $O(N/n)$.

Μόλις ολοκληρωθεί αυτή η φάση, σε κάθε επεξεργαστή έχει απομείνει ένας μόνο αριθμός. Το πρόβλημα πλέον είναι να προσθέσουμε n αριθμούς σε υπερκύβο με n κόμβους (δείτε το Σχ. 6.4). Από το Παράδειγμα 6.3 γνωρίζουμε όμως, ότι κάτι τέτοιο απαιτεί $2 \log n$ βήματα. Συνολικά λοιπόν, ο απαιτούμενος χρόνος θα διαμορφωθεί σε

$$T_n = O(N/n) + O(\log n) = O(N/n + \log n).$$

Το κόστος του νέου αλγορίθμου είναι $c_n = nT_n = O(N + n \log n)$. Ποιο είναι το συμπέρασμα; Όσο ο όρος N επικρατεί, δηλαδή $N = \Omega(n \log n)$, το κόστος είναι $c_n = O(N)$, δηλαδή βέλτιστο.

6.5 Όρια στις επιδόσεις—Νόμος Amdahl

Πολλές φορές κάποιο πρόγραμμά μας πετυχαίνει κάποια επιτάχυνση $s \leq n$. Προκειμένου να βελτιώσουμε παραπέρα την ταχύτητά του, είναι λογικό να αναρωτηθούμε εάν αυξάνοντας τον αριθμό των επεξεργαστών μπορούμε να επιτύχουμε καλύτερη επιτάχυνση. Μια απάντηση, δυστυχώς αρνητική, δίνει ο νόμος του Amdahl που θα εξετάσουμε σε αυτή την ενότητα.

Έστω ότι διαθέτουμε n επεξεργαστές και ότι το πρόγραμμά μας έχει τα εξής χαρακτηριστικά:

1. Ένα ποσοστό f των εντολών μπορεί να εκτελεστεί μόνο σειριακά.
2. Το υπόλοιπο ποσοστό $(1-f)$ των εντολών είναι πλήρως παραλληλοποιήσιμο.

Αυτή είναι μία λογική, αν και λίγο αισιόδοξη παραδοχή. Όντως, στην πράξη σχεδόν πάντα υπάρχουν τμήματα του προγράμματος που δεν μπορούν να παραλληλοποιηθούν καθόλου και τμήματα που μπορούν να παραλληλοποιηθούν (αν και ίσως όχι στον μέγιστο δυνατό βαθμό). Με βάση τις παραδοχές αυτές, ο χρόνος που απαιτείται για την εκτέλεση των μη παραλληλοποιήσιμων εντολών θα είναι επομένως fT_1, ενώ για τις υπόλοιπες εντολές με n επεξεργαστές θα απαιτηθεί χρόνος $(1-f)T_1/n$, αφού είναι πλήρως παραλληλοποιήσιμες. Συμπερασματικά,

$$T_n = fT_1 + (1-f)\frac{T_1}{n}$$

και άρα η επιτάχυνση θα είναι:

$$S_n = \frac{T_1}{T_n} = \frac{n}{nf + (1-f)} \leq \frac{1}{f}, \tag{6.4}$$

η οποία δίνει,

$$e_n = \frac{1}{nf + (1-f)}. \tag{6.5}$$

Οι σχέσεις (6.4) και (6.5) αποτελούν τον λεγόμενο *νόμο του Amdahl*, έναν από τους πιο απογοητευτικούς νόμους για την παράλληλη επεξεργασία. Ακόμα και αν μόνο το 1% του κώδικα είναι από τη φύση του σειριακός, δεν μπορούμε να επιτύχουμε επιτάχυνση πάνω από 100, όσους επεξεργαστές και να χρησιμοποιήσουμε. Ταυτόχρονα, καθώς ο αριθμός των επεξεργαστών μεγαλώνει, η αποδοτικότητα πέφτει κατά πολύ. Αν πάρετε το όριο καθώς το n τείνει στο άπειρο, θα δείτε ότι η αποδοτικότητα τείνει προς το μηδέν.

6.6 Καταρρίπτοντας τα όρια στις επιδόσεις

Ο νόμος του Amdahl θέτει περιορισμούς στην μέγιστη επίδοση που μπορούμε να επιτύχουμε προγραμματίζοντας παράλληλα. Όμως, υπάρχει μία λεπτομέρεια που μας διαφεύγει και

η οποία είναι σε θέση να μας επιτρέψει να ξεπεράσουμε το εμπόδιο του νόμου αυτού. Η λεπτομέρεια αυτή, καθώς και οι συνέπειές της εξετάζονται σε αυτή την ενότητα.

6.6.1 Το επιχείρημα

Είναι αλήθεια ότι, αν πάρουμε έναν συγκεκριμένο αλγόριθμο με συγκεκριμένο μέγεθος εισόδου N, δεν μπορούμε να τον παραλληλοποιήσουμε χρησιμοποιώντας οποιοδήποτε αριθμό επεξεργαστών. Για παράδειγμα, το πρόγραμμα ίσως να μπορεί να διασπαστεί σε 10 ανεξάρτητες εργασίες και άρα, να καταφέρουμε μια επιτάχυνση της τάξης του 10, χρησιμοποιώντας 10 επεξεργαστές. Από το σημείο αυτό και ύστερα, η προσθήκη ενός επιπλέον επεξεργαστή δεν πρόκειται να βελτιώσει την κατάσταση. Αντίθετα μάλιστα, αυξάνοντας τους επεξεργαστές μειώνουμε αυτόματα την αποδοτικότητα, αφού θα μένουν περισσότεροι επεξεργαστές άπραγοι. Στο Παράδειγμα 6.1, αν το μέγεθος του προβλήματος είναι 8 (εύρεση του αθροίσματος μεταξύ 8 στοιχείων), η χρήση π.χ. 16 ή περισσότερων επεξεργαστών είναι ανούσια για τον αλγόριθμο που δώσαμε. Ποια είναι επομένως, η λεπτομέρεια που μας διαφεύγει;

Η απάντηση βρίσκεται στο *μέγεθος της εισόδου* του αλγορίθμου. Στο παράδειγμά μας, αν επρόκειτο να βρούμε το άθροισμα 16 αριθμών, και όχι 8, θα είχαμε κάποιο όφελος από τη χρήση 16 επεξεργαστών. Με άλλα λόγια, *η χρήση επιπλέον επεξεργαστών δικαιολογείται αν προσπαθήσουμε να λύσουμε μεγαλύτερο πρόβλημα*. Κοιτάζοντάς το από την αντίθετη πλευρά, βλέπουμε ότι για έναν συγκεκριμένο αριθμό επεξεργαστών υπάρχει και κάποιο μέγεθος εισόδου για το πρόβλημα το οποίο τους εκμεταλλεύεται.

Έστω λοιπόν ότι, ένα ποσοστό f των εντολών είναι εκ φύσεως σειριακές. Αυτό που συμβαίνει στην πράξη είναι ότι το ποσοστό αυτό εξαρτάται από το μέγεθος εισόδου του προβλήματος και επομένως, θα είναι μία συνάρτηση $f(N)$. Κατ' επέκταση, ο χρόνος εκτέλεσης, η επιτάχυνση και η αποδοτικότητα είναι επίσης συναρτήσεις του μεγέθους εισόδου ($T_n(N)$, $S_n(N)$, κλπ.).

Επιπλέον, παρατηρήθηκε ότι σε πολλά προβλήματα το ποσοστό $f(N)$ *μειώνεται σταθερά*. Δηλαδή, τα αμιγώς σειριακά βήματα δεν αυξάνονται πολύ όταν αυξηθεί το μέγεθος του προβλήματος, σε αντίθεση με τους υπόλοιπους υπολογισμούς (οι οποίοι και μπορούν να παραλληλοποιηθούν) που αποτελούν ένα αυξανόμενο ποσοστό. Ως αποτέλεσμα, το ποσοστό $f(N)$ του σειριακού τμήματος μειώνεται με το N.

Οι παραπάνω σκέψεις δείχνουν ότι είναι δυνατή η βελτίωση στην επιτάχυνση και την αποδοτικότητα και αυτό αποτελεί επιχείρημα εναντίον του νόμου του Amdahl. Για κάθε αριθμό επεξεργαστών μπορούμε να βρούμε ένα μέγεθος εισόδου (δηλαδή ένα «αρκετά μεγάλο πρόβλημα»), το οποίο να μπορεί να επιτύχει μεγάλη επιτάχυνση και αποδοτικότητα. Άρα, η περίσσεια των επεξεργαστών μπορεί να είναι εκμεταλλεύσιμη, αρκεί να εκτελέσουμε ένα πρόβλημα με το κατάλληλο μέγεθος εισόδου.

6.6.2 Ο νόμος του Gustafson

Πολλές φορές ενδιαφερόμαστε να εκτελέσουμε το μεγαλύτερο πρόβλημα που μπορούμε χρησιμοποιώντας n επεξεργαστές αλλά με την προϋπόθεση ότι θα ολοκληρωθεί μέσα σε δεδομένο χρονικό διάστημα. Είναι πολύ συνηθισμένη τακτική στους χρήστες να εκμεταλλεύονται ένα πιο γρήγορο μηχάνημα όχι για να λύσουν ένα υπάρχον πρόβλημα γρηγορότερα, αλλά για να κατορθώσουν σε παρόμοιο χρόνο να λύσουν ένα μεγαλύτερο πρόβλημα. Για παράδειγμα, αν ενδιαφερόμαστε για πρόγνωση του καιρού της επόμενης μέρας, πρέπει να την ολοκληρώσουμε το πολύ σε 24 ώρες (αλλιώς δεν έχει κανένα νόημα). Προκειμένου να γίνει αυτό, οι αλγόριθμοι που χρησιμοποιούνται χειρίζονται σχετικά μικρό μέγεθος εισόδου, αφού οι υπολογισμοί είναι χρονοβόροι. Έτσι π.χ. θα εξετάσουν τις αλληλεπιδράσεις λίγων σχετικά σωμάτων της ατμόσφαιρας (βλ. το πρόβλημα των N σωμάτων στο Κεφάλαιο 1). Το αποτέλεσμα είναι ότι η πρόβλεψη δεν θα είναι ακριβής. Εάν μπορεί να χρησιμοποιηθεί ταχύτερος υπολογιστής, θα ενδιαφερθούμε να αυξήσουμε την ακρίβεια των υπολογισμών και όχι την ταχύτητά τους. Με άλλα λόγια, θα δώσουμε περισσότερα σώματα να εξεταστούν, αυξάνοντας έτσι το μέγεθος εισόδου του αλγορίθμου.

Έστω λοιπόν ότι ενδιαφερόμαστε ο χρόνος εκτέλεσης ενός προγράμματος να είναι $T_n(N)$ με n επεξεργαστές. Το ερώτημα είναι, ποια είναι η επιτάχυνση που πετυχαίνουμε; Όπως γνωρίζουμε,

$$S_n(N) = \frac{T_1(N)}{T_n(N)},$$

όπου όλα τα εμπλεκόμενα μεγέθη τα έχουμε εκφράσει ως συναρτήσεις του N.

Αν κατά την εκτέλεση *του παράλληλου προγράμματος* τα $f'(N)$ βήματα ήταν το ποσοστό του σειριακού κομματιού και το υπόλοιπο $1 - f'(N)$ εκτελέστηκε παράλληλα από τους n επεξεργαστές, τότε σε ένα σειριακό υπολογιστή θα χρειαζόμασταν χρόνο:

$$T_1(N) = f'(N)T_n(N) + (1 - f'(N))nT_n(N),$$

αφού η εργασία που εκτέλεσαν οι n επεξεργαστές παράλληλα σε χρόνο $(1 - f'(N))T_n(N)$ θα απαιτήσει n φορές περισσότερο χρόνο στον σειριακό υπολογιστή. Επομένως, βλέπουμε ότι:

$$S_n(N) = f'(N) + (1 - f'(N))n = n - (n - 1)f'(N). \tag{6.6}$$

Η σχέση (6.6) αποτελεί το *νόμο του Gustafson* και δείχνει ότι *μπορούμε να επιτύχουμε οποιαδήποτε επιτάχυνση και κατά συνέπεια οποιαδήποτε αποδοτικότητα «ρυθμίζοντας» κατάλληλα το $f'(N)$*. Η ρύθμιση αυτή φυσικά γίνεται μέσω του N, δηλαδή βρίσκοντας το κατάλληλο μέγεθος εισόδου το οποίο επιφέρει το επιθυμητό αποτέλεσμα. Έτσι, αν αυξηθεί ο αριθμός των επεξεργαστών (γίνει «κλιμάκωση» προς τα πάνω, όπως θα δούμε στην Ενότητα 6.8, του n) πρέπει να γίνει κλιμάκωση και του N, προκειμένου να επιτευχθεί η απαιτούμενη επιτάχυνση. Για τον λόγο αυτόν η σχέση (6.6) αναφέρεται πολλές φορές και ως *κλιμακωμένη επιτάχυνση* (scaled speedup).

Είναι σημαντικό να λεχθεί ότι ο νόμος του Gustafson δεν αντιβαίνει στον νόμο του Amdahl, ο οποίος προφανώς είναι σωστός. Απλά στον νόμο του Amdahl, υποτίθεται ότι το μέγεθος του προβλήματος μένει σταθερό και αυξάνονται μόνο οι επεξεργαστές. Αντίθετα, ο νόμος του Gustafson καλύπτει και την αύξηση στο μέγεθος του προβλήματος. Μπορούμε μάλιστα από τον έναν νόμο να μεταβούμε στον άλλο, όπως φαίνεται και στο Πρόβλημα 6.8, το οποίο δείχνει και κάποια λεπτά σημεία στην εφαρμογή των δύο νόμων.

6.7 Επιπρόσθετος χρόνος παραλληλισμού

Από τη μέχρι τώρα ανάλυση θα πρέπει να έχει διαπιστωθεί, ότι ο χρόνος εκτέλεσης ενός παράλληλου προγράμματος δεν περιλαμβάνει μόνο χρόνο, στον οποίο γίνονται οι καθαυτό υπολογισμοί, αλλά και επιπλέον χρόνο, που είναι απαραίτητος λόγω της ύπαρξης δύο ή παραπάνω επεξεργαστών. Αυτός ο επιπλέον χρόνος δεν εμφανίζεται κατά τη διάρκεια της εκτέλεσης σε σειριακό υπολογιστή, γι' αυτό και ονομάζεται *επιπρόσθετος χρόνος λόγω παραλληλισμού* (parallel overhead). Σε αυτή την ενότητα θα δούμε με λεπτομέρεια ποιοι παράγοντες συνεισφέρουν στον επιπρόσθετο αυτόν χρόνο.

Η βασική ερώτηση που έχει κανείς είναι γιατί η επιτάχυνση δεν είναι πάντα ίση με n; Δηλαδή, γιατί ο χρόνος εκτέλεσης ενός παράλληλου προγράμματος είναι συνήθως μεγαλύτερος του T_1/n, ενώ υπάρχουν n επεξεργαστές και λύνουν ταυτόχρονα το πρόβλημα; Δύο είναι οι σημαντικότεροι λόγοι: *οι επικοινωνίες* και *η ανισοκατανομή του φόρτου*.

Επικοινωνίες — Οι επικοινωνίες μεταξύ των επεξεργαστών είναι, ίσως, ο σημαντικότερος παράγοντας που εισάγει αναπόφευκτες καθυστερήσεις. Τις μελετήσαμε διεξοδικά στην Ενότητα 6.3. Έστω ότι κάθε επεξεργαστής σπαταλά κατά μέσο όρο χρόνο ίσο με t_{comm} για επικοινωνίες. Αθροιστικά, ο χρόνος που αφιέρωσαν όλοι οι επεξεργαστές για επικοινωνία θα είναι $T_{comm} = n \times t_{comm}$.

Παράδειγμα 6.6

» *Χρόνος επικοινωνίας στην πρόσθεση $n = N$ αριθμών στον υπερκύβο*

Όπως είδαμε στο Παράδειγμα 6.3, κάθε βήμα του αλγορίθμου περιείχε ένα βήμα πρόσθεσης και ένα βήμα επικοινωνίας, αφού οι εμπλεκόμενοι επεξεργαστές ήταν πάντα γειτονικοί. Συνολικά, $\log n$ βήματα σπαταλήθηκαν σε επικοινωνίες και επομένως, θα έχουμε $T_{comm} = n \log n$. Ο κάθε επεξεργαστής αφιέρωσε χρόνο $t_{comm} = T_{comm}/n = \log n$ για επικοινωνίες, κατά μέσο όρο.

Ανισοκατανομή φόρτου — Αν ένα πρόγραμμα χρειάστηκε χρόνο T_n, δεν σημαίνει ότι όλοι οι επεξεργαστές ήταν απασχολημένοι για όλες τις T_n χρονικές μονάδες (όταν λέμε «απα-

σχολημένος», εννοούμε ότι ένας επεξεργαστής είτε υπολόγιζε κάτι είτε επικοινωνούσε με κάποιον). Είναι πιθανόν, σε μία δεδομένη χρονική στιγμή, κάποιοι επεξεργαστές να ήταν απασχολημένοι, ενώ κάποιοι άλλοι να ήταν άπραγοι. Αυτό είναι το λεγόμενο πρόβλημα ανισοκατανομής φόρτου (load imbalance), όπου δεν εργάζονται όλοι οι επεξεργαστές για το ίδιο χρονικό διάστημα. Η κατάσταση αυτή είναι πολύ σημαντική αιτία επιπλέον καθυστερήσεων αφού, αν όλοι οι επεξεργαστές εργάζονταν για παρόμοιο χρόνο, η εκτέλεση του προγράμματος θα τελείωνε γρηγορότερα.

Όπως είδαμε στην Ενότητα 6.5, μία μορφή ανισοκατανομής φόρτου εμφανίζεται συχνά λόγω του ίδιου του αλγορίθμου. Στην περίπτωση αυτή, υπάρχουν τμήματα του κώδικα τα οποία δεν μπορούν να παραλληλοποιηθούν (είναι δηλαδή εκ φύσεως σειριακά). Μία τέτοια κατάσταση αναγκαστικά θα επιφέρει διαστήματα στα οποία μόνο ένας επεξεργαστής μπορεί να εργαστεί, ενώ οι υπόλοιποι απλώς περιμένουν.

Έστω λοιπόν, ότι κάθε επεξεργαστής ήταν κατά μέσο όρο άπραγος για χρόνο t_{idle}. Αθροιστικά ο συνολικός χρόνος που οι επεξεργαστές δεν έκαναν απολύτως τίποτε θα είναι ίσος με $T_{idle} = n \times t_{idle}$.

Παράδειγμα 6.7

» *Ανισοκατανομή φόρτου κατά την πρόσθεση n αριθμών*

Ο χρόνος για τον αλγόριθμο που δώσαμε στο Παράδειγμα 6.1, *αγνοώντας το χρονικό κόστος των επικοινωνιών*, είναι ίσος με $T_n = \log n$. Αθροιστικά, οι επεξεργαστές αφιέρωσαν χρόνο $n \times T_n = n \log n$ (ίσο με το κόστος του αλγορίθμου). Όμως, ο χρόνος που απαιτήθηκε για τους καθαυτό υπολογισμούς είναι $n-1$, αφού έγιναν ακριβώς $n-1$ προσθέσεις. Τα υπόλοιπα $T_{idle} = n \times T_n - (n-1) = n \log n - n + 1$ βήματα είναι αυτά που χάθηκαν λόγω ανισοκατανομής του φόρτου.

Συνοψίζοντας, είμαστε σε θέση να διατυπώσουμε και συμβολικά τις δύο βασικές πηγές καθυστερήσεων που υπεισέρχονται στον χρόνο παράλληλης εκτέλεσης. Οι δύο αυτές πηγές καθορίζουν τον επιπρόσθετο χρόνο λόγω παραλληλισμού, T_{ovh}, ο οποίος δίνεται από τη σχέση:

$$T_{ovh} = T_{comm} + T_{idle}.$$

Έστω λοιπόν ότι ο παράλληλος χρόνος εκτέλεσης ήταν ίσος με T_n. Αθροιστικά, για όλους τους επεξεργαστές, ο χρόνος που αφιερώθηκε για το πρόγραμμα ήταν nT_n, το οποίο είναι εξ ορισμού το κόστος του προγράμματος, c_n. Ο χρόνος που αφιερώθηκε σε καθαυτό υπολογισμούς, γνωστός και ως *χρόνος εργασίας*, W_n, προκύπτει από τον συνολικό χρόνο, αν αφαιρεθούν οι χρόνοι που σπαταλήθηκαν σε οτιδήποτε άλλο εκτός των καθαρών υπολογισμών, δηλαδή,

$$W_n = nT_n - T_{ovh}.$$

Προσέξτε ότι, εάν το πρόγραμμα είναι η παράλληλη εκδοχή του καλύτερου σειριακού προγράμματος για το ίδιο πρόβλημα, ο χρόνος εργασίας W_n πρέπει να είναι ίσος με τον σειριακό χρόνο εκτέλεσης, αφού αυτός περιλαμβάνει μόνο καθαρούς υπολογισμούς. Μπορούμε επομένως, να γράψουμε την τελευταία σχέση και ως $T_1 = nT_n - T_{ovh}$, η οποία δίνει:

$$T_n = \frac{T_1 + T_{ovh}}{n}. \tag{6.7}$$

Χρησιμοποιώντας τις σχέσεις (6.1), (6.2) και (6.7), βλέπουμε ότι η επιτάχυνση και η αποδοτικότητα ενός παράλληλου προγράμματος δίνονται από τις παρακάτω ισότητες:

$$S_n = \frac{T_1}{T_n} = n\frac{1}{1 + T_{ovh}/T_1},$$

$$e_n = \frac{S_n}{n} = \frac{1}{1 + T_{ovh}/T_1}, \tag{6.8}$$

όπου στον παρονομαστή εμφανίζεται ο λόγος του επιπρόσθετου χρόνου λόγω παραλληλισμού προς τον χρόνο των καθαυτό υπολογισμών. Όσο μεγαλύτερος είναι αυτός ο λόγος, τόσο μικρότερη είναι η επιτάχυνση και η αποδοτικότητα.

Μπορούμε επίσης να βρούμε την επίδραση του επιπρόσθετου χρόνου στο κόστος του παράλληλου προγράμματος. Από τις σχέσεις (6.3) και (6.7) προκύπτει ότι:

$$c_n = nT_n = T_1 + T_{ovh}.$$

Αν το πρόγραμμά μας απαιτείται να έχει βέλτιστο κόστος, όπως γνωρίζουμε, θα πρέπει ο λόγος c_n/T_1 να ισούται με $O(1)$. Επομένως, θα πρέπει

$$O(1) = \frac{c_n}{T_1} = 1 + \frac{T_{ovh}}{T_1},$$

που σημαίνει ότι για ένα πρόγραμμα με βέλτιστο κόστος, θα πρέπει να ισχύει

$$\frac{T_{ovh}}{T_1} = O(1), \tag{6.9}$$

δηλαδή, ο επιπρόσθετος χρόνος λόγω παραλληλισμού θα πρέπει να αυξάνει το πολύ γραμμικά σε σχέση με τον σειριακό χρόνο εκτέλεσης.

Παράδειγμα 6.8

» *Κόστος του αθροίσματος n αριθμών σε υπερκύβο με n επεξεργαστές*

Είδαμε ήδη (Παράδειγμα 6.3) ότι στον υπερκύβο χρειαζόμαστε χρόνο $T_n = 2\log n$, πράγμα που σημαίνει ότι θα πρέπει $T_{ovh} = nT_n - (n-1) = 2n\log n - n + 1$, αφού γνωρίζουμε ότι απαιτούνται συνολικά $T_1 = n-1$ προσθέσεις για να βρεθεί το άθροισμα των αριθμών. Το ίδιο αποτέλεσμα θα είχαμε (με πιο μεγάλη δυσκολία) αν υπολογίζαμε $T_{ovh} = T_{comm} + T_{idle}$. Ευτυχώς, οι απαιτούμενες ποσότητες είναι υπολογισμένες

(Παράδειγμα 6.6 και Παράδειγμα 6.7), οπότε βλέπουμε και με αυτόν τον τρόπο ότι $T_{ovh} = n \log n + n \log n - n + 1 = 2n \log n - n + 1$.

Με βάση αυτά, παρατηρούμε ότι $T_{ovh}/T_1 = O(n \log n)/O(n) = O(\log n)$. Ο αλγόριθμος δεν έχει βέλτιστο κόστος, αφού ο λόγος του επιπρόσθετου χρόνου παραλληλισμού προς τον σειριακό χρόνο αυξάνει με το n.

6.8 Κλιμάκωση

Σε αυτή την τελευταία ενότητα του κεφαλαίου, θα δούμε τι σημαίνει ο συχνά χρησιμοποιούμενος, αλλά όχι πάντα κατανοητός όρος «ικανότητα κλιμάκωσης». Εκτός από την ποιοτική της ερμηνεία, θα δούμε και ποσοτικά πώς καθορίζεται με βάση τους υπολογισμούς που κάναμε στις προηγούμενες ενότητες.

Πολύ συχνά στη βιβλιογραφία των παράλληλων υπολογιστών (σε όλο το φάσμα τους δηλαδή αρχιτεκτονική, αλγόριθμοι και προγραμματισμός) συναντάται ο όρος «κλιμάκωση» και διάφορα παράγωγά του. Η έννοια αυτή δεν έχει κάποιον αυστηρό ορισμό, αλλά συνήθως αντανακλά κάποια μεταβολή στις ιδιότητες που έχει ένα σύστημα. Συγκεκριμένα, *κλιμάκωση προς τα πάνω* (scaling up) είναι η αύξηση, ενώ *κλιμάκωση προς τα κάτω* (scaling down) είναι η μείωση σε κάποια παράμετρο ενός συστήματος. Ο απλός όρος *κλιμάκωση* (scaling) θα σημαίνει για εμάς την κλιμάκωση προς τα πάνω. Για παράδειγμα, όταν μελετούσαμε στην Ενότητα 6.5 τον νόμο του Amdahl, ουσιαστικά κάναμε μία πρόβλεψη για τη συμπεριφορά του αλγορίθμου, όταν γίνει κλιμάκωση στον αριθμό των επεξεργαστών.

Όπως όμως είδαμε, η κλιμάκωση επιφέρει αλλαγές στη συμπεριφορά ενός προγράμματος. Για παράδειγμα, στην Ενότητα 6.5 είδαμε ότι υπό ορισμένες προϋποθέσεις μειώνεται η αποδοτικότητα, ενώ στην Ενότητα 6.4 είδαμε ότι η κλιμάκωση προς τα κάτω επιφέρει ευεργετικές αλλαγές στο συνολικό χρόνο επικοινωνιών και το κόστος του προγράμματος. Η *ικανότητα κλιμάκωσης* (scalability) γενικά ορίζεται ως η ικανότητα ενός συστήματος να διατηρεί σταθερές κάποιες προκαθορισμένες ιδιότητές του, όταν γίνει κλιμάκωση (προς τα πάνω) κάποιων άλλων ιδιοτήτων. Συστήματα που διαθέτουν αυτή την ικανότητα θα ονομάζονται *συστήματα ικανά κλιμάκωσης* (scalable systems).

Στο πλαίσιο του παράλληλου προγραμματισμού ως «σύστημα» θεωρούμε τον συνδυασμό του αλγορίθμου και της αρχιτεκτονικής η οποία αναλαμβάνει την εκτέλεσή του. Οι ιδιότητες / χαρακτηριστικά που ενδιαφερόμαστε να κλιμακώσουμε είναι συνήθως ο αριθμός των επεξεργαστών ή το μέγεθος εισόδου του προγράμματος. Αυτό είναι λογικό, αφού π.χ. θέλουμε ένα πρόγραμμα που σχεδιάσαμε να μπορεί να εκτελεστεί, διατηρώντας τις καλές ιδιότητές του ακόμα και μετά από μία αναβάθμιση του υπολογιστή, όπως η προσθήκη περισσότερων επεξεργαστών.

Οι ιδιότητες που μας ενδιαφέρουν περισσότερο είναι η επιτάχυνση, η αποδοτικότητα και το κόστος ενός παράλληλου προγράμματος. Από αυτές, η ιδιότητα που προσδιορίζει την ικανότητα κλιμάκωσης είναι συνήθως η αποδοτικότητα του προγράμματος. Επομένως, για εμάς, ένα πρόγραμμα θα είναι ικανό κλιμάκωσης, αν μπορεί να διατηρεί την αποδοτικότητά του σταθερή όταν κλιμακώνεται ο αριθμός των επεξεργαστών. Όμως, όπως είδαμε στην Ενότητα 6.5, ο Amdahl ουσιαστικά απέδειξε ότι κανένα πρόγραμμα δεν μπορεί να είναι ικανό κλιμάκωσης αφού, όπως φάνηκε από τη σχέση (6.5), ακόμα και το παραμικρό ποσοστό σειριακών υπολογισμών είναι αρκετό να μειώνει την αποδοτικότητα, καθώς αυξάνεται ο αριθμός των επεξεργαστών (n).

Ο νόμος του Gustafson, όμως, ήρθε να αποδείξει ότι η αποδοτική χρήση των επιπλέον επεξεργαστών είναι εφικτή μόνο αν αυξηθεί κατάλληλα και το μέγεθος του προβλήματος. Επομένως, μια κλιμάκωση στον αριθμό των επεξεργαστών (n) πρέπει να συνοδεύεται και από αντίστοιχη κλιμάκωση στο μέγεθος εισόδου (N), προκειμένου να μη μειωθεί η αποδοτικότητα. Συνοψίζοντας λοιπόν τις παραπάνω παρατηρήσεις, μπορούμε πλέον να δώσουμε τον πλήρη ορισμό της ικανότητας κλιμάκωσης:

> Ένα παράλληλο σύστημα (αλγόριθμος + αρχιτεκτονική) είναι αν για κάθε κλιμάκωση του αριθμού επεξεργαστών υπάρχει κλιμάκωση του μεγέθους εισόδου, έτσι ώστε η αποδοτικότητα να παραμένει σταθερή.

Προσέξτε ότι, από τον ορισμό της αποδοτικότητας (σχέση (6.2)), έχουμε $S_n = ne_n$. Διατήρηση της αποδοτικότητας σε σταθερή τιμή σημαίνει ότι $S_n = O(n)$, δηλαδή αύξηση της επιτάχυνσης ανάλογη του n. Επομένως, η ικανότητα κλιμάκωσης αντανακλά επίσης την ικανότητα που έχει το σύστημα (μέσα από την κλιμάκωση του μεγέθους εισόδου) να αυξάνει την επιτάχυνσή του σε αναλογία με την αύξηση των επεξεργαστών. Με άλλα λόγια, δείχνει κατά πόσο το σύστημα είναι σε θέση να χρησιμοποιεί σωστά τους επιπλέον πόρους που του προσφέρονται.

Παράδειγμα 6.9

» Ικανότητα κλιμάκωσης για την πρόσθεση N αριθμών σε υπερκύβο n κόμβων

Ο αλγόριθμος στο Παράδειγμα 6.4 είχε χρόνο εκτέλεσης $T_n = O((N/n) \log n)$ και άρα αποδοτικότητα $e_n = O(1/\log n)$. Αν αυξηθεί ο αριθμός των επεξεργαστών, η αποδοτικότητα θα μειωθεί, ανεξάρτητα από οποιαδήποτε αύξηση στο μέγεθος εισόδου. Το σύστημα αυτό *δεν* είναι ικανό κλιμάκωσης.

Από την άλλη μεριά, ο αλγόριθμος στο Παράδειγμα 6.5 είχε χρόνο εκτέλεσης $T_n = O(N/n + \log n)$ και άρα αποδοτικότητα $e_n = O(N/(N + n \log n))$. Όσο ο όρος N επικρατεί στον παρονομαστή, δηλαδή $N = \Omega(n \log n)$, η αποδοτικότητα είναι $e_n = O(1)$, δηλαδή σταθερή. Επομένως, για κάθε αριθμό επεξεργαστών (n), υπάρχει ένα

μέγεθος εισόδου (N), έτσι ώστε η αποδοτικότητα να παραμένει σταθερή. Αυτό το σύστημα διαθέτει ικανότητα κλιμάκωσης.

Στο τελευταίο παράδειγμα, δεν είναι τυχαίο το γεγονός ότι ο δεύτερος αλγόριθμος είχε ικανότητα κλιμάκωσης. Ο λόγος είναι ότι υπάρχει άμεση σχέση μεταξύ ικανότητας κλιμάκωσης και βελτιστοποίησης του κόστους. Συγκεκριμένα, τα συστήματα που είναι ικανά κλιμάκωσης μπορούν πάντα να γίνουν βέλτιστου κόστους με κατάλληλη επιλογή των n και N. Αυτό μπορεί να φανεί αμέσως από τον ορισμό της αποδοτικότητας. Σταθερή αποδοτικότητα σημαίνει σταθερός λόγος T_1/nT_n, που αντίστοιχα σημαίνει σταθερός λόγος T_1/c_n, δηλαδή βέλτιστο κόστος.

Το τελευταίο σημείο που θα δούμε είναι το πόσο *εύκολα* κλιμακώνεται ένα παράλληλο σύστημα. Γενικά, ένα σύστημα θεωρείται ότι έχει *χαμηλή ικανότητα κλιμάκωσης* (poorly scalable), αν μικρή αύξηση των επεξεργαστών απαιτεί μεγάλη αύξηση στο μέγεθος εισόδου του προβλήματος, προκειμένου να διατηρηθεί σταθερή η αποδοτικότητα. Αντίθετα, έχει *υψηλή ικανότητα κλιμάκωσης* (highly scalable), αν μικρή αύξηση των επεξεργαστών απαιτεί και μικρή αύξηση στο μέγεθος εισόδου.

Για να δούμε ποσοτικά αυτές τις έννοιες, θα ανατρέξουμε στη σχέση (6.8). Για να είναι σταθερή η αποδοτικότητα σε κάποια τιμή E, θα πρέπει ο λόγος T_{ovh}/T_1 του επιπρόσθετου χρόνου λόγω παραλληλισμού δια του σειριακού χρόνου εκτέλεσης να είναι σταθερός. Αναδιοργανώνοντας τη σχέση (6.8), βλέπουμε ότι θα πρέπει να ισχύει:

$$T_1 = \frac{E}{1-E} T_{ovh}.$$

Ο σειριακός χρόνος εκτέλεσης είναι συνάρτηση μόνο του μεγέθους εισόδου, αφού είναι χρόνος καθαυτό υπολογισμών. Από την άλλη, ο επιπρόσθετος χρόνος παραλληλισμού είναι συνάρτηση τόσο του μεγέθους εισόδου όσο και του αριθμού των επεξεργαστών. Επομένως, μπορούμε να ξαναγράψουμε την τελευταία εξίσωση ως εξής:

$$T_1(N) = \frac{E}{1-E} T_{ovh}(n, N). \tag{6.10}$$

Η (6.10) σχετίζει τον αριθμό των επεξεργαστών με το μέγεθος εισόδου του προβλήματος προκειμένου η αποδοτικότητα να έχει μία σταθερή τιμή E. Από αυτή τη σχέση μπορούμε να αποφανθούμε για την ευκολία στην ικανότητα κλιμάκωσης ενός συστήματος. Σε γενικές γραμμές, την υψηλότερη ικανότητα κλιμάκωσης έχουμε όταν το N είναι γραμμική συνάρτηση του n, ενώ αν το N είναι εκθετική συνάρτηση του n, το σύστημα έχει πολύ χαμηλή ικανότητα κλιμάκωσης.

6.9 Σύνοψη

Στο κεφάλαιο αυτό είδαμε τις βασικές μετρικές που προσδιορίζουν τις επιδόσεις ενός παράλληλου προγράμματος, αλλά και τους παράγοντες που επηρεάζουν άμεσα αυτές τις μετρικές. Ο σειριακός χρόνος εκτέλεσης, ο οποίος είναι συνάρτηση του μεγέθους εισόδου N του προβλήματος, είναι ο χρόνος που απαιτείται για τους καθαρούς υπολογισμούς και μερικές φορές συναντάται και ως το «έργο» του προβλήματος. Το ίδιο έργο πρέπει να φέρει εις πέρας και το παράλληλο πρόγραμμα με μόνη διαφορά ότι πρέπει να συμπληρωθεί από επιπρόσθετο (και άχρηστο από πλευράς υπολογισμών) έργο, που οφείλεται κυρίως σε δύο παράγοντες:

- την αναγκαιότητα των επικοινωνιών μεταξύ επεξεργαστών,
- την ανισοκατανομή φόρτου στους επεξεργαστές.

Αν όλοι οι επεξεργαστές εργάζονταν για το ίδιο χρονικό διάστημα και οι χρόνοι επικοινωνίας ήταν αμελητέοι, τότε το παράλληλο πρόγραμμα θα μπορούσε να εκτελεστεί n φορές γρηγορότερα από το σειριακό, κάνοντας χρήση n επεξεργαστών. Όμως αυτό δεν συμβαίνει πάντα στην πράξη. Έτσι, η επιτάχυνση, που ορίζεται ως ο σειριακός χρόνος δια τον παράλληλο χρόνο εκτέλεσης, είναι γενικά μικρότερη της ιδεώδους (n). Με την ίδια λογική, ο λόγος της επιτάχυνσης δια του αριθμού των επεξεργαστών, που δίνει την αποδοτικότητα του παράλληλου προγράμματος, είναι γενικά μικρότερος του 100%.

Το κόστος του παράλληλου προγράμματος περιλαμβάνει και το έργο των καθαυτό υπολογισμών αλλά και το αναγκαίο επιπρόσθετο έργο λόγω των παραπάνω παραγόντων. Όσο το επιπρόσθετο αυτό έργο δεν ξεπερνά, σε μέγεθος, το έργο του προβλήματος τότε το πρόγραμμα λέγεται ότι έχει βέλτιστο κόστος. Οι επιπρόσθετες καθυστερήσεις που εισάγονται στην παράλληλη εκτέλεση μπορούν να μειωθούν με την αύξηση του κόκκου παραλληλίας (η οποία μπορεί να γίνει με την τεχνική των εικονικών επεξεργαστών). Είδαμε μάλιστα περιπτώσεις όπου η μείωση είναι τέτοια, ώστε ένα πρόγραμμα να αποκτά βέλτιστο κόστος όταν εκτελείται σε μικρότερο αριθμό επεξεργαστών.

Όμως, μας ενδιαφέρει και η περίπτωση όπου ο αριθμός των επεξεργαστών αυξάνεται, πάντα με την προοπτική να κάνουμε την εκτέλεση ταχύτερη. Από μόνη της αυτή η αύξηση συνήθως δεν είναι ικανή να βελτιώσει την αποδοτικότητα, αντίθετα την μειώνει σταθερά, όπως προβλέπει ο νόμος του Amdahl. Αν όμως η αύξηση στον αριθμό των επεξεργαστών συνδυαστεί με αύξηση στο μέγεθος εισόδου (δηλαδή αν χρησιμοποιήσουμε το ισχυρότερο σύστημα για λύση μεγαλύτερου προβλήματος), τότε με βάση το νόμο του Gustafson είναι πολλές φορές δυνατή η επίτευξη των επιθυμητών επιδόσεων. Τα συστήματα που το καταφέρνουν αυτό, και πιο συγκεκριμένα, τα συστήματα τα οποία μέσω κατάλληλης αύξησης του μεγέθους εισόδου μπορούν να διατηρούν την αποδοτικότητά τους ψηλά όταν αυξάνεται ο αριθμός των επεξεργαστών, ονομάζονται ικανά κλιμάκωσης. Η κλιμακω-

σιμότητα, πάντως, ως έννοια είναι δύσκολη και σχετικά αόριστη [Hill90].

Ο νόμος του Amdahl διατυπώθηκε το 1967 [Amda67] και αποτέλεσε ένα αποτέλεσμα / επιχείρημα με πολύ μεγάλη επιρροή στην ιστορία των παράλληλων υπολογιστών. Η «απάντηση» του Gustafson ήρθε 21 χρόνια αργότερα [Gust88] και αναζωπύρωσε την έρευνα γύρω από τα παράλληλα συστήματα και τις επιδόσεις τους. Μια σύγχρονη εκδοχή του νόμου του Amdahl που ταιριάζει καλύτερα στα σημερινά πολυπύρηνα συστήματα διατυπώθηκε από τους Hill και Marty [HiMa08]. Η αντίστοιχη απάντηση δόθηκε από τους [SuCh10].

Τα βιβλία που ασχολούνται με παράλληλα συστήματα αναφέρονται, έστω και σύντομα, σε κάποια από τα θέματα που συζητήσαμε στο κεφάλαιο αυτό. Η εκτενέστερη, ίσως, παρουσίαση περιλαμβάνεται στο [GGKK03], αν και λίγο παλαιότερο.

Προβλήματα

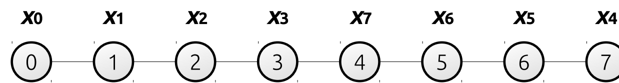
6.1 – Σχεδιάστε έναν αλγόριθμο για την εύρεση του μέγιστου μεταξύ n αριθμών χρησιμοποιώντας n επεξεργαστές (το n είναι δύναμη του 2, όπως στο Παράδειγμα 6.1 του κεφαλαίου). Ποιος είναι ο χρόνος εκτέλεσης και ποια η επιτάχυνση;

6.2 – Έχουμε n αριθμούς, $\{x_i \mid i = 0, 1, \ldots, n-1\}$, όπου το n είναι δύναμη του δύο, όπως στο Παράδειγμα 6.1 και θέλουμε να υπολογίσουμε όλα τα αθροίσματα:

$$\begin{aligned}
\Sigma_0 &= x_0 \\
\Sigma_1 &= x_0 + x_1 \\
\Sigma_2 &= x_0 + x_1 + x_2 \\
&\vdots \\
\Sigma_{n-1} &= x_0 + x_1 + \cdots + x_{n-1}
\end{aligned}$$

Το πρόβλημα αυτό είναι γνωστό ως *προθεματικά αθροίσματα* (prefix sums). Στο Παράδειγμα 6.1 υπολογίσαμε μόνο το Σ_{n-1}. Δώστε έναν αλγόριθμο για το πρόβλημα αυτό χρησιμοποιώντας n επεξεργαστές, έτσι ώστε τελικά ο επεξεργαστής i να έχει το άθροισμα Σ_i. Υπολογίστε επίσης τον χρόνο εκτέλεσης και την επιτάχυνση. Προσοχή: *υποτίθεται ότι ένας επεξεργαστής μπορεί σε κάποιο βήμα να στείλει τον αριθμό που διαθέτει μόνο σε έναν από τους άλλους επεξεργαστές.*

Υπόδειξη: Η άσκηση αυτή δεν είναι εύκολη. Ξεκινήστε από το Παράδειγμα 6.1 για $n = 8$. Κατόπιν, προσπαθήστε ταυτόχρονα με τις υπάρχουσες μεταφορές και προσθέσεις



Σχήμα 6.5 *Αντιστοιχιση στον γραμμικό γράφο για το Πρόβλημα 6.5*

στο Σχ. 6.1 να παρεμβάλετε νέες μεταφορές και προσθέσεις, ώστε στον επεξεργαστή 1 να καταλήξει το Σ_1. Μετά, κάνετε αντίστοιχες προσθήκες για το Σ_2 κ.ο.κ. Αφού ολοκληρώσετε, θα φανεί η γενική λύση για οποιοδήποτε n.

6.3 – Να βρεθεί η αποδοτικότητα και το κόστος για τους αλγορίθμους που σχεδιάσατε στο Πρόβλημα 6.1 και στο Πρόβλημα 6.2.

6.4 – Να βρεθεί μία σχέση μεταξύ κόστους και αποδοτικότητας και μία σχέση μεταξύ κόστους και επιτάχυνσης.

6.5 – Όπως στο Παράδειγμα 6.2, σας ζητείται να προσθέσετε αριθμούς σε έναν γραμμικό γράφο. Η μόνη διαφορά είναι η αντιστοίχιση, η οποία φαίνεται στο Σχ. 6.5 για 8 κόμβους. Υπολογίστε τον χρόνο που απαιτείται για την πρόσθεση σε αυτό το δίκτυο. Τι παρατηρείτε; Είναι η νέα αντιστοίχιση καλύτερη; Γενικά, σε έναν γραμμικό γράφο n κόμβων και εφαρμόζοντας τη νέα αντιστοίχιση, τι χρόνος απαιτείται για την πρόσθεση;

6.6 – Σχεδιάστε τις καμπύλες της αποδοτικότητας και της επιτάχυνσης που δίνει ο νόμος του Amdahl συναρτήσει του n, για $f = 0.01$, 0.02, 0.05 και 0.1.

6.7 – Ο νόμος του Amdahl προβλέπει ότι καθώς το n τείνει στο άπειρο, η αποδοτικότητα τείνει προς το μηδέν. Συμβαίνει αυτό για ένα πρόγραμμα που έχει βέλτιστο κόστος για κάθε n;

6.8 – Κατά την εκτέλεση ενός προγράμματος σε έναν παράλληλο υπολογιστή με 10 επεξεργαστές βρέθηκε από μετρήσεις ότι, για ένα ποσοστό του χρόνου $f_G = 50\%$ η εκτέλεση ήταν σειριακή (δηλαδή μόνο ένας επεξεργαστής εργαζόταν), ενώ στο υπόλοιπο $1 - f_G = 50\%$ είχαμε πλήρως παράλληλη εκτέλεση (εργάζονταν όλοι). Με βάση αυτά τα ποσοστά ο προγραμματιστής G έβγαλε το συμπέρασμα, χρησιμοποιώντας τον νόμο του Gustafson, ότι η επιτάχυνση ήταν:

$$S_{10} = 10 - (10 - 1) \times 0.5 = 5.5,$$

ενώ ο προγραμματιστής A χρησιμοποίησε τον νόμο του Amdahl και βρήκε ότι:

$$S_{10} = \frac{10}{10 \times 0.5 + (1 - 0.5)} = 1.82.$$

Δεδομένου ότι και οι δύο νόμοι είναι σωστοί, ποιος έκανε το λάθος;

6.9 – Στο Παράδειγμα 6.5 είδαμε έναν αλγόριθμο για την πρόσθεση N αριθμών σε ένα υπερκύβο με $n < N$ κόμβους. Όπως κάναμε στα Παραδείγματα 6.6–6.8, προσπαθήστε

να βρείτε τον επιπλέον χρόνο λόγω παραλληλισμού και δείτε αν πληρούται η σχέση (6.9), ώστε να διαπιστώσετε και με αυτόν τον τρόπο ότι ο αλγόριθμος μπορεί να έχει βέλτιστο κόστος.

6.10 – Χρησιμοποιώντας την εξίσωση (6.10) και τα αποτελέσματα από το Πρόβλημα 6.9, βρείτε τη σχέση του μεγέθους εισόδου και του αριθμού των επεξεργαστών, ώστε ο αλγόριθμος στο Παράδειγμα 6.5 να έχει σταθερή αποδοτικότητα 75%. Η ικανότητα κλιμάκωσης του αλγορίθμου είναι υψηλή ή χαμηλή;

Βιβλιογραφία

[**Adap14a**] Adapteva, *Parallella Reference Manual*, Sept. 2014.

[**Adap14b**] Adapteva, *Epiphany Architecture Reference*, Nov. 2014.

[**AMD13**] Advanced Micro Devices Inc, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, May 2013.

[**Amda67**] G.M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *Proc. AFIPS '67, Spring Joint Computer Conference*, Atlantic City, NJ, pp. 483–485, 1967.

[**BDM09**] G.G. Blake, R.G. Dreslinski and T. Mudge, "A Survey of Multicore Processors", *IEEE Signal Processing Magazine*, Vol. 26, 11 2009.

[**Bute97**] D.R. Butenhof, *Programming With POSIX Threads*, Addison-Wesley, 1997.

[**CJP07**] B. Chapman, G. Jost and R. van der Pas, *Using OpenMP: Protable Shared Memory Parallel Programming*, MIT Press, 2007.

[**CKDL10**] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor", *IEEE Micro*, Vol. 30, No. 2, pp. 16–29, Mar. 2010.

[**CMDK01**] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan and J. McDonald, *Parallel Programming in OpenMP*, Morgan Kaufmann, 2001.

[**CoHu07**] P. Conway and B. Hughes, "The AMD Opteron Northbridge Architecture", *IEEE Micro*, Vol. 27, No. 2, pp. 10–21, Mar. 2007.

[CSG99] D.E. Culler, J.P. Singh and A. Gupta, *Parallel Computer Architecture: A Hardware / Software Approach*, Morgan Kaufmann, San Fransisco, CA, 1999.

[DaTo04] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann Publishers, 2004.

[Dima13] V.V. Dimakopoulos, "Parallel Programming Models", *Smart Multicore Embedded Systems*, M. Torquati, K. Bertels, S. Karlsson and F. Pacull, eds., Springer Science+Business Media, pp. 3–20, 2013.

[DYN03] J. Duato, S. Yalamanchili and L. Ni, *Interconnection Networks: an Engineering Approach*, Morgan Kaufmann Publishers, 2003.

[Flyn72] M.J. Flynn, "Some computer organizations and their effectiveness", *IEEE Transactions on Computers*, Vol. C-21, pp. 948–960, 1972.

[FWM94] G.C. Fox, R.D. Williams and P.C. Messina, *Parallel Computing Works!*, Morgan Kaufmann, 1994.

[GGKK03] A. Grama, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing, 2nd Ed.*, Addison Wesley, San Fransisco, CA, 2003.

[GHTL14] W. Gropp, T. Hoefler, R. Thakur and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface, 3rd Ed.*, MIT Press, 2014.

[GLS14] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming With the Message-Passing Interface, 3rd Ed.*, MIT Press, 2014.

[Gust88] J.L. Gustafson, "Reevaluating Amdahl's Law", *Communications of the ACM*, Vol. 31, No. 5, pp. 532-533, May 1988.

[Hill90] M.D. Hill, "What is scalability?", *SIGARCH Computer Architecture News*, Vol. 18, No. 4, pp. 18–21, Dec. 1990.

[Hill98] M.D. Hill, "Multiprocessors should support simple memory consistency models", *IEEE Computer*, Vol. 31, No. 8, pp. 28–34, Aug. 1998.

[HiMa08] M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era", *IEEE Computer*, Vol. 41, No. 7, pp. 33–38, July 2008.

[Hors12] H.M. and Horst Gietl, "Supercomputers – Prestige Objects or Crucial Tools for Science and Industry?", *Software Development Practice*, Vol. 1, No. 1, pp. 63–79, Nov. 2012.

[IEEE00] IEEE, "Standard for Information Technology – Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment J: Advanced Real-time Extensions, POSIX.1c, Threads extensions", IEEE Std 1003.1j-2000, 2000.

[IEEE95] IEEE, "Standard for Information Technology – Portable Operating System Interface (POSIX) - POSIX.1c, Threads extensions", IEEE Std 1003.1c-1995, 1995.

[Inte09] Intel Corporation, *An Introduction to the Intel QuickPath Interconnect*, Jan. 2009.

[Inte15a] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Jan. 2015.

[Inte15b] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, Jan. 2015.

[KKLR00] K. Kavi, H.-S. Kim, B. Lee, A. R and Hurson, "Shared memory and distributed shared memory systems: A survey", *Advances in Computers: Emphasizing Distributed Systems*, M.V. Zelkowits, ed., Elsevier, pp. 55–108, 2000.

[Lamp79] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocessor programs", *IEEE Transactions on Computers*, pp. 690–691, Sept. 1979.

[Leig92] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, 1992.

[LiSn08] C. Lin and L. Snyder, *Principles of Parallel Programming*, Pearson, 2008.

[Marg07] A. Μάργαρης, *MPI : Θεωρία και εφαρμογές*, Τζιόλας, 2007.

[MPI13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.3*, May 2008.
» http://www.mpi-forum.org/docs/mpi-1.3/

[MPI22] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*, HLRS Stuttgart, Sept. 2009.
» http://www.mpi-forum.org/docs/mpi-2.2/

[MPI31] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*, June 2015.
» http://www.mpi-forum.org/docs/mpi-3.1/

[MRR12] M. McCool, J. Reinders and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, 2012.

[MSM04] T.G. Mattson, B.A. Sanders and B.L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley Professional, 2004.

[NiCh11] M.A. Nielsen and I.L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*, Cambridge University Press, 2011.

[OMP25] OpenMP ARB, "OpenMP Application Program Interface V2.5", May 2005.
» http://www.openmp.org/mp-documents/spec25.pdf

[OMP30] OpenMP ARB, "OpenMP Application Program Interface V3.0", May 2008.
» http://www.openmp.org/mp-documents/spec30.pdf

[OMP31] OpenMP ARB, "OpenMP Application Program Interface V3.1", July 2011.
» http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[OMP40] OpenMP ARB, "OpenMP Application Program Interface V4.0", July 2013.
» http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[Pach11] P. Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011.

[PaDi12] Σ. Παπαδάκης and Κ. Διαμαντάρας, *Προγραμματισμός και αρχιτεκτονική συστημάτων παράλληλης επεξεργασίας*, Εκδόσεις Κλειδάριθμος, 2012.

[Patt98] A. Pattavina, *Switching Theory: Architecture and Performance in Broadband ATM Networks*, John Wiley and Sons Ltd., 1998.

[PMT13] Γ. Πάντζιου, Β. Μάμαλης and Α. Τομαράς, *Εισαγωγή στον παράλληλο υπολογισμό: Πρότυπα, αλγόριθμοι, προγραμματισμός*, Εκδόσεις Νέων Τεχνολογιών, 2013.

[Quin03] M. Quinn, *Parallel Programming in C With MPI and OpenMP*, McGraw-Hill, 2003.

[SHW11] D. Sorin, M.D. Hill and D.A. Wood, *A Primer on Memory Consistency and Cache Coherence*, Morgan and Claypool Publishers, 2011.

[StRa13] W.R. Stevens and S.A. Rago, *Advanced Programming in the UNIX Environment, 3rd Edition*, Addison-Wesley Professional, 2013.

[SuCh10] X.-H. Sun and Y. Chen, "Reevaluating Amdahl's law in the multicore era", *Journal of Parallel and Distributed Computing*, Vol. 70, No. 2, pp. 183–188, Feb. 2010.

[SuLa05] H. Sutter and J. Larus, "Software and the Concurrency Revolution", *ACM Queue*, Vol. 3, No. 7, pp. 54–62, Sept. 2005.

[T500] Top500 Supercomputer Sites, http://www.top500.org.

[WiAl04] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2nd Ed.*, Pearson, 2004.

[WMFC15] M.D. Wael, S. Marr, B.D. Fraine, T.V. Cutsem and W.D. Meuter, "Partitioned Global Address Space Languages", *ACM Comput. Surv.*, Vol. 47, No. 4, pp. 1–27, July 2015.

[Xu10] J. Xu, *Topological Structure and Analysis of Interconnection Networks*, Springer Publishing Company, 2010.

Ευρετήριο

Amdahl's law, 234

barrier, *δες* κλήση φραγής
Benes, *δες* δίκτυο › Benes
block scheduling, *δες* δρομολόγηση › τμηματική
blocking network, *δες* δίκτυο › εμποδιστικό

cache coherency, *δες* συνοχή κρυφής μνήμης
chained directory protocol, *δες* πρωτόκολλο καταλόγων › αλυσιδωτών
circuit switching, *δες* μεταγωγή › κυκλώματος
clusters, *δες* παράλληλοι υπολογιστές › συστάδες
communicator, 214
control parallelism, *δες* προγραμματιστικό μοντέλο › παραλληλισμός ελέγχου
critical section, *δες* κρίσιμη περιοχή
crossbar switch, *δες* διασταυρωτικός διακόπτης

data parallelism, *δες* προγραμματιστικό μοντέλο › παραλληλισμός δεδομένων
directory-based protocol, *δες* πρωτόκολλο καταλόγων, *δες* πρωτόκολλο καταλόγων
distributed shared memory (DSM), *δες* κατανεμημένη κοινή μνήμη

efficiency, *δες* αποδοτικότητα
execution entities, *δες* οντότητες εκτέλεσης

fork-join, 121
full-map directory protocol, *δες* πρωτόκολλο καταλόγων › πλήρων

Gray code, *δες* κώδικας Gray
Gustafson's law, 236

highly scalable, *δες* ικανότητα κλιμάκωσης › υψηλή
hypercube, *δες* δίκτυο › υπερκύβος

initial thread, *δες* νήμα › αρχικό
irregular parallelism, *δες* παραλληλισμός › ακανόνιστος

limited directory protocol, *δες* πρωτόκολλο καταλόγων › περιορισμένων
lock, *δες* κλειδαριά

main thread, *δες* νήμα › αρχικό
mapping, *δες* αντιστοίχιση

memory barrier, *δες* φράχτης μνήμης
memory consistency, *δες* συνέπεια μνήμης
mesh, *δες* δίκτυο › πλέγμα
MESI snooping protocol, *δες* πρωτόκολλο παρακολούθησης › MESI
message-passing, *δες* προγραμματιστικό μοντέλο › μεταβίβαση μηνυμάτων
MIMD, *δες* παράλληλοι υπολογιστές
MPI, 180, 218
multicore processors, *δες* πολυπύρηνοι επεξεργαστές
multistage networks, *δες* δίκτυο › πολλαπλών επιπέδων
mutual exclusion, *δες* αμοιβαίος αποκλεισμός

nested parallelism, *δες* παραλληλισμός › εμφωλευμένος
non-blocking network, *δες* δίκτυο › μη εμποδιστικό
non-uniform memory accessy (NUMA), *δες* ανομοιόμορφη προσπέλαση μνήμης

orchestration, *δες* ενορχήστρωση
oversubscription, *δες* υπερεγγραφή

packet switching, *δες* μεταγωγή › store-and-forward
parallel overhead, *δες* επιπρόσθετος χρόνος παραλληλισμού
partitioning, *δες* διαχωρισμός
placement, *δες* τοποθέτηση
poorly scalable, *δες* ικανότητα κλιμάκωσης › χαμηλή
process, *δες* διεργασία

rearrangeable network, *δες* δίκτυο › επαναδιατάξιμο
recirculating network, *δες* δίκτυο › επανακυκλοφορίας
reduction, *δες* υποβίβαση
remote memory access (RMA), *δες* απομακρυσμένη προσπέλαση μνήμης
router, *δες* διαδρομητής
routing, *δες* διαδρόμηση

scalabillity, *δες* κλιμάκωση › ικανότητα
scalable multiprocessors, *δες* παράλληλοι υπολογιστές › κλιμακώσιμοι
scalable parallel system, *δες* ικανότητα κλιμάκωσης › παράλληλο σύστημα
scalable system, *δες* ικανότητα κλιμάκωσης › σύστημα
scaled speedup, *δες* επιτάχυνση › κλιμακωμένη
scaling, *δες* κλιμάκωση
scheduling, *δες* δρομολόγηση
self-scheduling, *δες* αυτοδρομολόγηση
sequential consistency, *δες* συνέπεια μνήμης › ακολουθιακή
shared address space, *δες* προγραμματιστικό μοντέλο › κοινόχρηστος χώρος διευθύνσεων
shared variables model, *δες* προγραμματιστικό μοντέλο › κοινόχρηστων μεταβλητών
SIMD, *δες* παράλληλοι υπολογιστές › SIMD
snooping protocol, *δες* πρωτόκολλο παρακολούθησης
speedup, *δες* επιτάχυνση
SPMD, 19, 182
superlinear speedup, *δες* επιτάχυνση › υπεργραμμική
switching, *δες* μεταγωγή
symmetric mulutiprocessor, *δες* συμμετρικοί πολυεπεξεργαστές
synchronization, *δες* συγχρονισμός

thread, *δες* νήμα
torus, *δες* δίκτυο › torus

uniform memory access (UMA), *δες* ομοιό-

μορφη προσπέλαση μνήμης
universal network, *δες* δίκτυο › μη εμποδιστικό
virtual cut-through, *δες* μεταγωγή › virtual cut-through
virtual processor, *δες* εικονικός επεξεργαστής
wormhole switching, *δες* μεταγωγή › wormhole
write-invalidate protocol, *δες* πρωτόκολλο παρακολούθησης › εγγραφής-ακύρωσης
write-update protocol, *δες* πρωτόκολλο παρακολούθησης › εγγραφής-ενημέρωσης

αμοιβαίος αποκλεισμός, 123, 128–131
ανάθεση, 20
ανομοιόμορφη προσπέλαση μνήμης (NUMA), 108
αντιστοίχιση, 21, 22
αποδοτικότητα, 225
απομακρυσμένη προσπέλαση μνήμης (RMA), 216
αυτοδρομολόγηση, 151
δίκτυο
  baseline, 37
  Benes, 37
  torus, 83
  Ωμέγα, 37
  εμποδιστικό, 37
  επαναδιατάξιμο, 37
  επανακυκλοφορίας, 38
  μη εποδιστικό (παγκόσμιο), 37
  πλέγμα, 83
  πολλαπλών επιπέδων, 32
  συμμετρικό Δέλτα, 32
  υπερκύβος, 83
διάσπαση, 20, 22–24
  αναδρομική, 23
  ελεγκτή-εργάτη, 23
  επαναληπτική, 23
διαδρομητής, 77
διαδρόμηση, 94
  e-cube, 96
διασταυρωτικός διακόπτης, 31
διαχωρισμός, 20
διεργασία, 119
δρομολόγηση, 141
  dynamic, 164
  guided, 164
  runtime, 165
  static, 164
  αυτοδρομολόγηση, 151
  με άλματα, 144
  τμηματική, 141
εικονικός επεξεργαστής, 229
ενορχήστρωση, 21
επεξεργαστές πίνακα, *δες* παράλληλοι υπολογιστές › SIMD
επικοινωνία
  ασύγχρονη, 187
  διασκόρπιση, 199, 201
  εκπομπή, 199, 200
  μη-εμποδιστική, 208
  μηδενικών αντιγραφών, 210
  μονόπλευρη, 216
  πολλαπλή εκπομπή, 199, 202
  συλλογή, 199, 201
  συλλογική, 198–203
  σύγχρονη, 187
  υποβίβαση, 202
επιπρόσθετος χρόνος παραλληλισμού, 237
επιτάχυνση, 223
  κλιμακωμένη, 236
  υπεργραμμική, 224
ικανότητα κλιμάκωσης
  παράλληλο σύστημα, 241
  σύστημα, 240

υψηλή, 242
χαμηλή, 242
κατανεμημένη κοινή μνήμη, 19, 107
κλήση φραγής, 135–136
κλειδαριά, 128
κλιμάκωση, 240
ικανότητα, 240
κλιμακώσιμο σύστημα, *δες* ικανότητα κλιμάκωσης › σύστημα
κοινόχρηστος χώρος διευθύνσεων, *δες* προγραμματιστικό μοντέλο › κοινόχρηστος χώρος διευθύνσεων
κρίσιμη περιοχή, 128
κόκκος παραλληλίας
λεπτός, 9
χονδρός, 9
κόκκος παραλληλίας › χονδρός, 232
κόστος, 226
βέλτιστο, 226
κώδικας Gray, 87
μερικός, 89
λεπτός κόκκος παραλληλίας, *δες* κόκκος παραλληλίας › λεπτός
μαζικά παράλληλοι υπολογιστές, *δες* παράλληλοι υπολογιστές › μαζικά παράλληλοι
μεταβίβαση μηνυμάτων, *δες* προγραμματιστικό μοντέλο › μεταβίβαση μηνυμάτων
μεταγωγή, 96
store-and-forward, 96
virtual cut-through, 97
wormhole, 97
κυκλώματος, 97
μοντέλο von Neumann, 4
νήμα, 119
αρχικό, 119, 125
κύριο, *δες* νήμα › αρχικό
νόμος του
Amdahl, 234
Gustafson, 236
ομοιόμορφη προσπέλαση μνήμης (UMA), 70
οντότητες εκτέλεσης, 120

παράλληλοι υπολογιστές
SIMD, 14
κατανεμημένης μνήμης, 16, 75–115
κλιμακώσιμοι, 109
κοινόχρηστης μνήμης, 16, 27–73
μαζικά παράλληλοι, 8
ομαδοποιημένοι, 16, 104
συμμετρικοί, 30
συστάδες, 105
παράλληλος προγραμματισμός
άμεσος, 18
έμμεσος, 18
μοντέλο, 18–20
παραλληλισμός
ακανόνιστος, 167
δεδομένων, *δες* προγραμματιστικό μοντέλο › παραλληλισμός δεδομένων
εμφωλευμένος , 166
πολυεπεξεργαστές, *δες* παράλληλοι υπολογιστές
πολυπύρηνοι επεξεργαστές, 7, 66–70, 110–113
πολυϋπολογιστές, *δες* παράλληλοι υπολογιστές › κατανεμημένης μνήμης
προγραμματιστικό μοντέλο
κοινόχρηστος χώρος διευθύνσεων, 19, 117–177
κοινόχρηστων μεταβλητών, 117
μεταβίβαση μηνυμάτων, 19, 179–220
παραλληλισμός δεδομένων, 18
παραλληλισμός ελέγχου, 19
πρωτόκολλο καταλόγων, 46, 52
αλυσιδωτών, 56–58

περιορισμένων, 54–56
πλήρων, 53–54
πρωτόκολλο παρακολούθησης, 46
MESI, 51
εγγραφής-ακύρωσης, 47
εγγραφής-ενημέρωσης, 47
συγχρονισμός, 124, 131–136
συνέπεια μνήμης, 59
TSO, 64
ακολουθιακή, 60
αποδέσμευσης, 65
ασθενούς σειράς, 65
επεξεργαστή, 64
χαλαρής σειράς, 66
συνθήκες ανταγωνισμού, 131
συνοχή κρυφής μνήμης, 45
συστάδες, *δες* παράλληλοι υπολογιστές › συστάδες
τοποθέτηση, 20, 21
υπερεγγραφή, 141
υποβίβαση, 157
φράχτης μνήμης, 63
χονδρός κόκκος παραλληλίας, *δες* κόκκος παραλληλίας › χονδρός