ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ **ΜΕΣΟΛΟΓΓΙΟΥ**
ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ ΚΑΙ ΟΙΚΟΝΟΜΙΑΣ
ΤΜΗΜΑ ΕΦΑΡΜΟΓΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΣΤΗΝ ΔΙΟΙΚΗΣΗ ΚΑΙ ΤΗΝ ΟΙΚΟΝΟΜΙΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΓΟΡΙΘΜΟΙ ΣΕ ΕΙΚΟΝΙΚΗ ΠΑΡΑΛΛΗΛΗ ΜΗΧΑΝΗ ΜΕ ΧΡΗΣΗ ΤΟΥ ΠΡΩΤΟΚΟΛΛΟΥ MPI (MESSAGE PASSING INTERFACE)

Επιβλέπων καθηγητής: **Χρήστος Ευθυμιόπουλος**

Σπουδάστριες: **Ελευθερία Δημογέροντα**
Μαρία Σωτήρχου

ΜΕΣΟΛΟΓΓΙ 2007

ΠΡΟΛΟΓΟΣ

Λόγω της αλματώδους ανάπτυξης του κλάδου της Τεχνολογίας και της Πληροφορικής, η Επιστήμη των υπολογιστών, μετά από 40 χρόνια σχεδόν πλήρους ενασχόλησης με το σειριακό προγραμματισμό, άρχισε να αναγνωρίζει τη σημασία του παράλληλου προγραμματισμού. Τα τελευταία χρόνια, η ανάγκη για αύξηση της ταχύτητας των υπολογιστών οδήγησε στην ανάπτυξη της παράλληλης επεξεργασίας, ως ενός νέου τρόπου επίτευξης καλύτερης απόδοσης. Έτσι, άρχισε να ωριμάζει η ιδέα της συνεργασίας πολλών επεξεργαστών για την επίτευξη ενός κοινού στόχου (δηλαδή την λύση ενός προβλήματος σε μικρότερο χρόνο). Έχουμε φτάσει πλέον σε ένα ικανοποιητικό επίπεδο κόστους, τέτοιο ώστε ακόμα και μία μικρή σχετικά επιχείρηση να μπορεί να προμηθευτεί ένα παράλληλο σύστημα και να επωφεληθεί από την χρήση του.

Στην παρούσα εργασία, θα μελετήσουμε τα παράλληλα συστήματα κατανεμημένης και διαμοιραζόμενης μνήμης. Θα παρουσιάσουμε ένα από τα βασικότερα πρωτόκολλα επικοινωνίας, το MPI (Message Passing Interface), το οποίο συναντάμε στα συστήματα κατανεμημένης μνήμης. Γίνεται αναφορά στα χαρακτηριστικά και τις λειτουργίες του. Η μελέτη μας ολοκληρώνεται με την υλοποίηση κώδικα σε σειριακά και παράλληλα συστήματα (cluster), με τη χρήση του πρωτοκόλλου MPI, για ορισμένους γνωστούς αλγόριθμους των υπολογιστικών μαθηματικών. Σε κάθε παράδειγμα συγκρίνεται η απόδοση του σειριακού με τον παράλληλο αλγόριθμο.

Θα θέλαμε να ευχαριστήσουμε θερμά τον επιβλέποντα καθηγητή μας κ. Χρήστο Ευθυμιόπουλο, για την υποστήριξή του καθώς επίσης και για τις εύστοχες παρατηρήσεις και υποδείξεις του καθ'όλη τη διάρκεια εκπόνησης της εργασίας αυτής. Η διεκπεραίωση της διπλωματικής έγινε στις εγκαταστάσεις του Κέντρου Ερευνών Αστρονομίας και Εφαρμοσμένων Μαθηματικών (Κ.Ε.Α.Ε.Μ.) της Ακαδημίας Αθηνών. Ιδιαίτερες ευχαριστίες οφείλουμε στους γονείς μας καθώς και σε όλους όσοι με τον δικό τους τρόπο συνετέλεσαν στην εκπόνηση της εργασίας αυτής.

Περιεχόμενα

1. ΕΙΣΑΓΩΓΗ

1.1 Ιστορικά στοιχεία

Η ανάγκη για αύξηση της ταχύτητας των υπολογισμών έχει οδηγήσει τα τελευταία χρόνια στην ανάπτυξη ενός νέου τρόπου επίτευξης καλύτερης απόδοσης των υπολογιστικών συστημάτων, *την παράλληλη επεξεργασία.*

Ο όρος παράλληλη επεξεργασία σημαίνει ότι πολλοί υπολογιστικοί κόμβοι (επεξεργαστές) συνεργάζονται για την γρηγορότερη επίλυση ενός προβλήματος-αλγορίθμου. Η ανάπτυξη της παράλληλης επεξεργασίας προέκυψε από το γεγονός ότι ο ρυθμός αύξησης της απόδοσης των απλών υπολογιστικών συστημάτων ακολουθεί φθίνουσα πορεία λόγω φυσικών περιορισμών, όπως η πεπερασμένη ταχύτητα μετάδοσης των σημάτων (Hwang και Briggs [1988]).

Το 1945 ο πρώτος ηλεκτρονικός επεξεργαστής στον υπολογιστή ENIAC μπορούσε να εκτελέσει 1000 αριθμητικές πράξεις ανά δευτερόλεπτο. Σήμερα στους συμβατικούς υπολογιστές RISC εκτελούνται 100.000.000 αριθμητικές πράξεις ανά δευτερόλεπτο. Η αύξηση της ισχύος των υπολογιστών αναμένεται να συνεχιστεί, αλλά με μικρότερους ρυθμούς σε σχέση με το παρελθόν. Για να αυξηθεί η υπολογιστική ισχύς θα πρέπει να σχεδιαστούν και να κατασκευαστούν ειδικοί σειριακοί επεξεργαστές με την χρήση μεγάλου αριθμού κυκλωμάτων VLSI (Very large scale integration). Ωστόσο, η κατασκευή αυτών των επεξεργαστών παραμένει δαπανηρή.

Πριν από μια δεκαετία, οι κατασκευαστές επεξεργαστών αποφάσισαν ότι είναι πιο οικονομική η σύνδεση πολλών απλών επεξεργαστών VLSI σε ένα υπολογιστή από το να σχεδιάσουν και να κατασκευάσουν έναν ειδικό επεξεργαστή για ειδικούς σκοπούς. Για παράδειγμα, αν κάποιος χρειάζεται δέκα φορές περισσότερη υπολογιστική ισχύ θα χρησιμοποιήσει δέκα υπολογιστικές μονάδες. Η ανάγκη για υπολογιστική ισχύ εισήγαγε σε ευρεία βάση τα συστήματα διαμοιραζόμενης μνήμης.

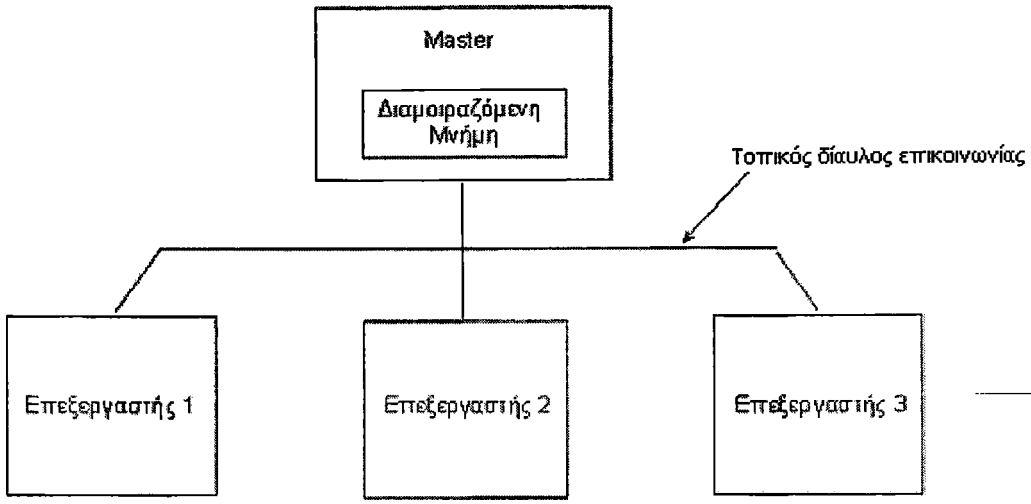
Η γρήγορη ανάπτυξη νέας γενιάς ταχύτερων σειριακών επεξεργαστών επέτρεψε τη σύνδεση πολλών τέτοιων επεξεργαστών, έτσι ώστε η υπολογιστική ισχύς να φτάσει στο απαιτούμενο επίπεδο, και να είναι δυνατή η επίλυση πολύπλοκων προβλημάτων. Έτσι η παράλληλη επεξεργασία δεν συγκρούεται με την σειριακή αλλά συνεργάζονται αρμονικά στο υπολογιστικό σύστημα.

Η ανάπτυξη των επεξεργαστών VLSI και η ταχύτατη εξέλιξη της τεχνολογίας κυκλωμάτων θα δημιουργήσουν ταχύτερα και πιο δυνατά παράλληλα συστήματα. Μεγάλες εταιρείες που παράγουν παράλληλα συστήματα συνεχίζουν να ενσωματώνουν τους τελευταίας γενιάς πανίσχυρους σειριακούς επεξεργαστές στα συστήματα τους.

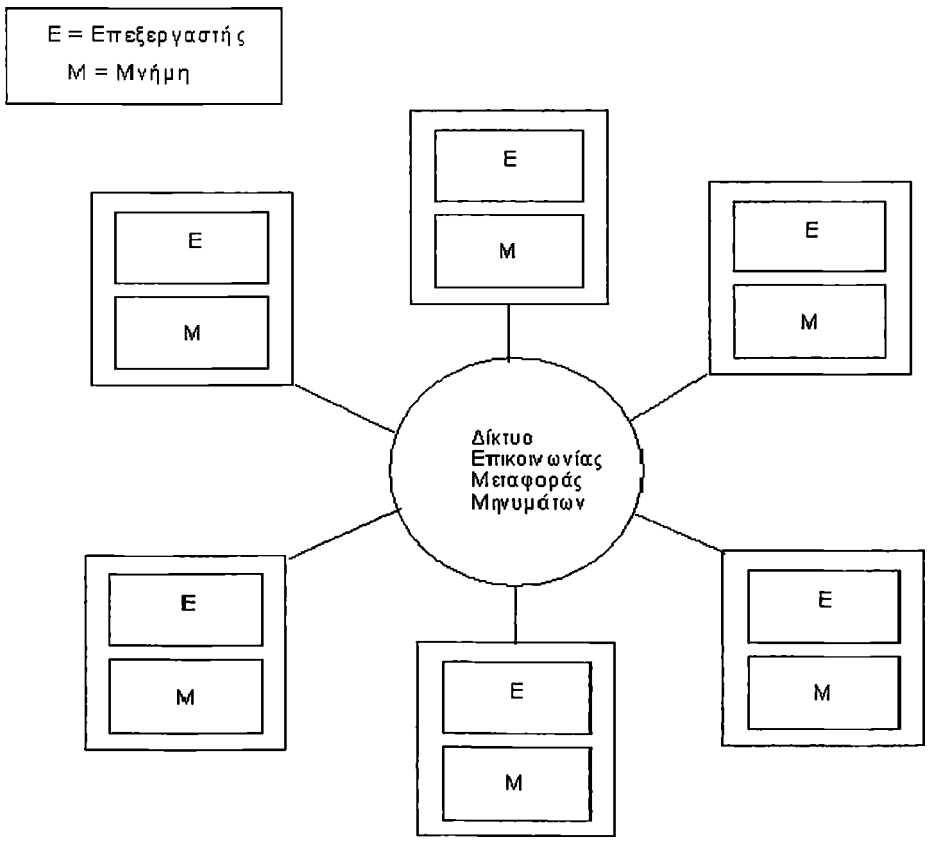
1.2 Κατηγορίες παράλληλων συστημάτων

Τα παράλληλα συστήματα διακρίνονται σε δύο βασικές κατηγορίες: τα συστήματα *κατανεμημένης μνήμης* και τα συστήματα *διαμοιραζόμενης μνήμης*. Στα συστήματα διαμοιραζόμενης μνήμης όλοι οι υπολογιστές μοιράζονται μια μνήμη αλλά κάθε υπολογιστής έχει δικό του επεξεργαστή (Σχήμα 1.1). Η επικοινωνία γίνεται διαμέσου του κοινού τοπικού διαύλου (local bus) που συνδέει την μνήμη με τους επεξεργαστές. Στα συστήματα κατανεμημένης μνήμης ο κάθε υπολογιστής λειτουργεί ως μεμονωμένο σύστημα που έχει δικό

του επεξεργαστή και δική του μνήμη. Η επικοινωνία μεταξύ των υπολογιστών γίνεται δικτυακά (Σχήμα 1.2). Παρακάτω θα δούμε αναλυτικά την λειτουργία καθεμίας εκ των δύο κατηγοριών παράλληλων συστημάτων.



Σχήμα 1.1 - Οργάνωση Συστήματος Διαμοιραζόμενης Μνήμης



Σχήμα 1.2 - Οργάνωση Συστήματος Κατανεμημένης Μνήμης (Hwang και Briggs [1988])

1.2.1 Συστήματα με διαμοιραζόμενη μνήμη (shared memory)

Σε ένα υπολογιστικό σύστημα που χρησιμοποιεί έναν επεξεργαστή σημαντικό ρόλο παίζει το ταίριασμα της ταχύτητας προσπέλασης της μνήμης με την ταχύτητα του επεξεργαστή. Για να βελτιωθεί η μέση απόδοση της μνήμης χρησιμοποιούμε διάφορους μηχανισμούς, όπως η κρυφή μνήμη (cache) και οι καταχωρητές της κεντρικής μονάδας επεξεργασίας. Τέτοιοι μηχανισμοί αξιοποιούν το πλεονέκτημα της τυπικότητας στην πρόσβαση της μνήμης, έτσι ώστε να μειώσουν το μέσο χρόνο προσπέλασης της κύριας μνήμης (Hockney και Jesshope [1984]).

Κατά παρόμοιο τρόπο, στα παράλληλα συστήματα, η ταχύτητα προσπέλασης της μνήμης πρέπει να ισορροπεί με την ταχύτητα των επεξεργαστών για να ξεπερνιέται το πρόβλημα καθυστέρησης που δημιουργείται από συνεχή πρόσβαση στη μνήμη. Όταν «διαβάζουμε» τη μνήμη του επεξεργαστή πολλές φορές ή και ταυτόχρονα τότε η μνήμη «χάνει» από την ταχύτητα της με αποτέλεσμα να επιβραδύνεται η προσπέλαση της.

Ένα υπολογιστικό σύστημα αποτελείται από έναν επεξεργαστή και την μνήμη του, τα οποία συνδέονται μεταξύ τους με έναν δίαυλο. Ο δίαυλος είναι το μονοπάτι από όπου «περνούν» τα δεδομένα από και προς τη μνήμη. Η παραπάνω αρχιτεκτονική επεκτείνεται στην παράλληλη επεξεργασία δεδομένων.

Στην παράλληλη αρχιτεκτονική έχουμε συνήθως έναν κεντρικό επεξεργαστή, που ονομάζεται κύριος επεξεργαστής (master processor). Ο κύριος επεξεργαστής διαθέτει απευθείας πρόσβαση στη μια και μοναδική μνήμη του συστήματος, που στην περίπτωση αυτή ονομάζεται διαμοιραζόμενη μνήμη. Επίσης μπορούμε να συνδέσουμε με τον master από 1-10 ή και περισσότερους επεξεργαστές με την βοήθεια διαύλου. Οι επεξεργαστές αυτοί ονομάζονται σκλάβοι (slaves) και μέσω του κεντρικού διαύλου που ενώνει τον master με τους slaves αποκτούν πρόσβαση στην μια και μοναδική μνήμη.

Για να είναι πιο επιτυχής η παράλληλη επεξεργασία δεδομένων θα πρέπει να έχουμε συνδέσει επεξεργαστές της ίδιας τεχνολογίας ή τουλάχιστον, αν αυτό δεν είναι εφικτό, παρόμοιας ταχύτητας. Αυτό είναι το πιο σημαντικό κριτήριο για την δημιουργία ενός συστήματος διαμοιραζόμενης μνήμης, γιατί ο δίαυλος και η κενή μνήμη μπορούν να εκτελέσουν μια μόνο αίτηση για επεξεργασία από έναν επεξεργαστή κάθε φορά.

Για να γίνει η προσπέλαση μιας θέσης μνήμης από έναν επεξεργαστή θα πρέπει να χρησιμοποιηθεί ο δίαυλος για μικρό χρονικό διάστημα. Εδώ μπορούμε να κατανοήσουμε το λόγο για τον οποίο οι επεξεργαστές πρέπει να είναι της ίδιας τεχνολογίας. Σε αντίθετη περίπτωση ένας συγκριτικά πιο «αργός» επεξεργαστής θα χρησιμοποιούσε περισσότερη ώρα το δίαυλο με αποτέλεσμα να καθυστερούν οι υπόλοιποι επεξεργαστές και να αναιρούνται στην πράξη τα οφέλη από την παράλληλη επεξεργασία δεδομένων.

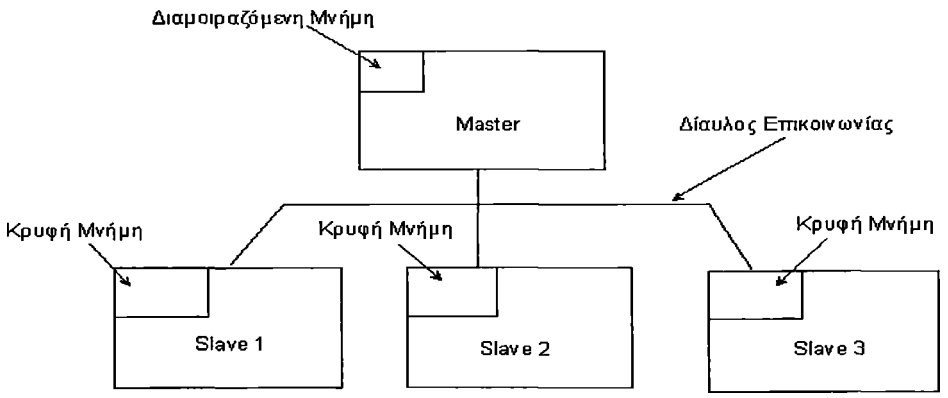
Χρήσιμο είναι να μελετήσουμε και την ταχύτητα του διαύλου που θα χρησιμοποιήσουμε για το σύστημα διαμοιραζόμενης μνήμης. Ανάλογα με τη συχνότητα του διαύλου καθορίζεται το πλήθος των επεξεργαστών που μπορούν να λειτουργήσουν αποδοτικά. Συνήθως χρησιμοποιούμε το ελάχιστο πλήθος επεξεργαστών από όσους θα μπορούσαμε να συνδέσουμε. Για παράδειγμα, αν η συχνότητα του διαύλου είναι 50MHz και ο μέσος ρυθμός προσπέλασης του επεξεργαστή στην μνήμη είναι 5MHz, ο δίαυλος θα κορεστεί στους 10 επεξεργαστές.

Παρόλο που η συχνότητα του διαύλου είναι μεγαλύτερη από την ταχύτητα προσπέλασης του επεξεργαστή στη μνήμη παραμένει συχνά ο κίνδυνος να εμφανιστεί το πρόβλημα του ανταγωνισμού πρόσβασης στη μνήμη. Για την αποφυγή του παραπάνω προβλήματος, τα σύγχρονα εμπορικά συστήματα διαμοιραζόμενης μνήμης, έχουν συνήθως 20-30 επεξεργαστές, που είναι ο βέλτιστος αριθμός επεξεργαστών που αντιστοιχεί στους περιορισμούς που επιβάλλονται από τις σημερινές τυπικές τιμές ταχύτητας του διαύλου (Hockney και Jesshope [1984]). Σ' ένα τέτοιο σύστημα υπάρχει μια μοναδική διαμοιραζόμενη μνήμη με ένα κοινό δίαυλο που συνδέει όλους τους επεξεργαστές. Ο κάθε επεξεργαστής διαθέτει περαιτέρω τη δική του, κρυφή ιδιωτική μνήμη στην οποία μπορεί να αποθηκεύει τις πρόσφατα χρησιμοποιούμενες τιμές μεταβλητών μιας εφαρμογής. Έτσι, ελέγχει πρώτα τη δική του τοπική μνήμη για να δει αν οι ανάγκες της καταχώρησης βρίσκονται εκεί και μόνο σε περίπτωση που δε βρίσκονται εκεί τα

στοιχεία της καταχώρησης ο επεξεργαστής προσπαθεί να προσπελάσει την κύρια διαμοιραζόμενη μνήμη με την βοήθεια του διαύλου. Η χρήση τοπικής κρυφής μνήμης βοηθά στη μείωση του ανταγωνισμού πρόσβασης στην μνήμη. Έτσι επιτρέπεται η πρόσβαση στην μνήμη από πιο πολλούς επεξεργαστές συνδεδεμένους στο δίαυλο χωρίς ο τελευταίος να υπερφορτώνεται.

Κάθε κρυφή μνήμη είναι ένα μικρό τμήμα μνήμης με υψηλή ταχύτητα και περιέχει ζεύγη της μορφής «διεύθυνση – δεδομένα». Σε κάθε αναφορά μνήμης από έναν επεξεργαστή, η διεύθυνση της μνήμης αναζητείται πρώτα στην τοπική μνήμη. Σε περίπτωση που η αναζητούμενη μονάδα δεδομένων δεν βρεθεί εκεί, χρησιμοποιείται ο δίαυλος για να προσπελαστεί περαιτέρω η διαμοιραζόμενη μνήμη. Την στιγμή που η μονάδα δεδομένων διαβάζεται από την διαμοιραζόμενη μνήμη, αυτόματα τοποθετείται ένα αντίγραφό της στην τοπική μνήμη, ώστε να είναι διαθέσιμη για τις επόμενες προσπελάσεις από τον συγκεκριμένο επεξεργαστή.

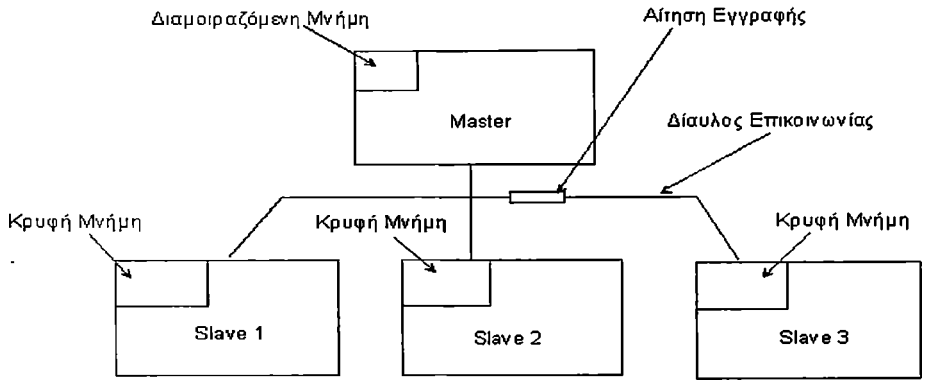
Ένα πολύ ενδιαφέρον θέμα για την κρυφή μνήμη στα συστήματα διαμοιραζόμενης μνήμης είναι ότι πολλά αντίγραφα από το ίδιο τμήμα της κύριας μνήμης μπορεί να είναι αποθηκευμένα σε πολλές διαφορετικές μνήμες (Σχήμα 1.3). Για παράδειγμα, ας υποθέσουμε ότι ο Επεξεργαστής 1 διαβάζει τη διεύθυνση μνήμης 1000 από την διαμοιραζόμενη μνήμη. Ύστερα δημιουργείται ένα αντίγραφο στην τοπική κρυφή μνήμη του Επεξεργαστή 1. Αν και οι υπόλοιποι Επεξεργαστές, 2 και 3, διαβάσουν την ίδια διεύθυνση μνήμης από τη διαμοιραζόμενη μνήμη, τότε θα αποθηκευτεί και στις δικές τους τοπικές μνήμες ένα τέτοιο αντίγραφο. Έτσι, θα υπάρχουν τέσσερα αντίγραφα αυτής της θέσης μνήμης (1000). Όσο οι προσπελάσεις σε αυτή την θέση μνήμης γίνονται μόνο για την ανάγνωση της θέσης αυτής δεν υπάρχει κανένα πρόβλημα από την ύπαρξη πολλαπλών αντιγράφων. Έτσι καθένας από τους επεξεργαστές χρησιμοποιώντας το δικό του αντίγραφο από την τοπική του μνήμη μπορεί να εκτελεί λειτουργία ανάγνωσης χωρίς να χρειάζεται να προσπελάσει τη διαμοιραζόμενη μνήμη. Ωστόσο, αν κάποιος επεξεργαστής επιχειρήσει να κάνει μια λειτουργία εγγραφής στην θέση μνήμης 1000, θα δημιουργηθεί πρόβλημα. Αφού υπάρχουν πολλαπλά αντίγραφα του ιδίου τμήματος μνήμης, θα πρέπει τα αντίγραφα αυτά να έχουν μόνιμα το ίδιο περιεχόμενο. Επομένως, δεν είναι αποδεκτό να γίνει εγγραφή σε ένα αντίγραφο της μνήμης.



Σχήμα 1.3 – Σύστημα διαμοιραζόμενης μνήμης

Αυτό το πρόβλημα χειρισμού πολλών αντιγράφων από το ίδιο τμήμα μνήμης καλείται πρόβλημα συνοχής της κρυφής μνήμης (cache coherence). Διάφορες λύσεις έχουν υλοποιηθεί σε εμπορικά και πειραματικά συστήματα διαμοιραζόμενης μνήμης. Ας σημειωθεί ωστόσο ότι κάθε τέτοια λύση επιβαρύνει το κόστος (υλικού και χρόνου) στο σύστημα, και επομένως υποβαθμίζει την απόδοσή του. Για τα συστήματα που λειτουργούν με διαύλους, μια λύση του

προβλήματος είναι να εκπέμπεται στο δίαυλο κάθε αίτηση εγγραφής από κάθε επεξεργαστή (Σχήμα 1.4). Έτσι κάθε κρυφή μνήμη παρακολουθεί τις αιτήσεις που κυκλοφορούν στο δίαυλο για να δει αν κάποιες δικές του καταχωρήσεις, έχουν εγγραφεί στην μνήμη από κάποιον άλλο επεξεργαστή. Έτσι όταν κάτι αλλάζει στην θέση μνήμης που όλοι οι επεξεργαστές έχουν αντίγραφό της, οι επεξεργαστές θα προσθέτουν τις αλλαγές και έτσι θα είναι όλα τα αντίγραφα πάντα ίδια. Αν οι κρυφές μνήμες χρησιμοποιήσουν αυτήν την τακτική, κάθε λειτουργία εγγραφής θα πηγαίνει άμεσα από το δίαυλο στην κύρια μνήμη (Hockney και Jesshope [1984]).



Σχήμα 1.4 – Λύση προβλήματος συνοχής μνήμης (cache coherence)

1.2.2 Συστήματα με κατανεμημένη μνήμη (distributed memory)

Μια διαφορετική προσέγγιση για τη μείωση του ανταγωνισμού των τμημάτων μνήμης στα συστήματα παράλληλης επεξεργασίας αποτελούν τα παράλληλα συστήματα κατανεμημένης μνήμης (Σχήμα 1.2). Τα συστήματα αυτά περιλαμβάνουν μια αρκετά μεγάλη τοπική μνήμη για κάθε επεξεργαστή, καθώς και ένα δίκτυο επικοινωνίας για την αλληλεπίδραση των επεξεργαστών μέσω ενός μηχανισμού μεταφοράς μηνυμάτων. Κάθε επεξεργαστής μπορεί να προσπελάσει απευθείας την ιδιωτική του μνήμη, λειτουργεί, δηλαδή, αυτόνομα χρησιμοποιώντας τα δεδομένα που είναι αποθηκευμένα στο δικό του τμήμα τοπικής μνήμης. Επίσης μπορεί να στέλνει και να λαμβάνει δεδομένα από και προς οποιονδήποτε άλλον επεξεργαστή, χρησιμοποιώντας το δίκτυο επικοινωνίας ανταλλαγής μηνυμάτων. Έτσι, αν ο επεξεργαστής i χρειάζεται να προσπελάσει κάποιο δεδομένο από τη μνήμη του επεξεργαστή j, θα απευθύνει αίτημα προς τον j μέσω δικτυακού μηνύματος. Ο j, πάλι μέσω δικτυακού μηνύματος θα ανταποκριθεί στο αίτημα αποστέλλοντας το αιτούμενο δεδομένο στον επεξεργαστή i (Σχήμα 1.2).

Η βασική οργάνωση του παράλληλου συστήματος κατανεμημένης μνήμης είναι διαφορετική από την οργάνωση του συστήματος διαμοιραζόμενης μνήμης και απαιτεί ένα διαφορετικό εννοιολογικό μοντέλο προγραμματισμού για την ανάπτυξη του λογισμικού. Στα συστήματα κατανεμημένης μνήμης, ο επεξεργαστής γνωρίζει τη διαφορά ανάμεσα στα τοπικά και απομακρυσμένα τμήματα μνήμης. Κάθε ζευγάρι επεξεργαστής-μνήμη συμπεριφέρεται σαν ένα μικρό, αυτόνομο σύστημα υπολογιστή. Ο επεξεργαστής μπορεί να διαβάζει και να γράφει δεδομένα ελεύθερα, χρησιμοποιώντας την δική του τοπική μνήμη. Κοινά δεδομένα θα πρέπει είτε να αντιγραφούν σε όλες τις ιδιωτικές μνήμες είτε να φυλάσσονται από έναν επεξεργαστή, ο οποίος θα τα διαμοιράζει κατόπιν αιτήσεων. Όταν οι επεξεργαστές θέλουν να ανταλλάξουν

δεδομένα, αυτό πρέπει να γίνει μέσω μιας ρητής ενέργειας ανταλλαγής μηνυμάτων χρησιμοποιώντας το δίκτυο επικοινωνίας.

Στα παράλληλα συστήματα κατανεμημένης μνήμης έχει αναπτυχθεί μια μεγάλη ποικιλία διαφορετικών τοπολογιών δικτύου επικοινωνίας με σκοπό τη μείωση του κόστους και της πολυπλοκότητας του δικτύου, με ταυτόχρονη αύξηση της ταχύτητας επικοινωνίας μεταξύ των επεξεργαστών.

Ο προγραμματισμός των συστημάτων κατανεμημένης μνήμης περιλαμβάνει ένα επιπλέον επίπεδο πολυπλοκότητας σε σύγκριση με τον προγραμματισμό των συστημάτων διαμοιραζόμενης μνήμης. Οι προγραμματιστές πρέπει να γνωρίζουν την αρχιτεκτονική της μηχανής και να σχεδιάζουν τα προγράμματά τους έτσι ώστε να ελαχιστοποιούν τη σχετική επιβάρυνση εξαιτίας των χρονοβόρων καθυστερήσεων επικοινωνίας μέσω του δικτύου. Πιο συγκεκριμένα, η κατανεμημένη μνήμη απαιτεί την οργάνωση του προγράμματος έτσι ώστε κάθε διεργασία να στηρίζεται κυρίως στις δικές της τοπικές μεταβλητές (Hwang και Briggs [1988]).

Ο παράλληλος προγραμματισμός με *ανταλλαγή μηνυμάτων* είναι σχεδιασμένος έτσι ώστε να παρέχει μεγαλύτερη αποδοτικότητα στην εκτέλεση προγραμμάτων σε συστήματα κατανεμημένης μνήμης. Στην ανταλλαγή μηνυμάτων η οργάνωση του προγράμματος αντικατοπτρίζει την υποκείμενη οργάνωση του συστήματος κατανεμημένης μνήμης. Κάθε διεργασία δημιουργείται από μια κλήση διαδικασίας με όλα τα απαιτούμενα διαμοιραζόμενα δεδομένα που περνιούνται σαν παράμετροι τιμών στη διαδικασία. Από τη στιγμή που μια διεργασία έχει δημιουργηθεί βασίζεται κυρίως στις δικές της τοπικές μεταβλητές, που βρίσκονται αποθηκευμένες στην τοπική μνήμη του επεξεργαστή. Όλη η επικοινωνία των διεργασιών πραγματοποιείται μέσα από ανταλλαγή μηνυμάτων χρησιμοποιώντας θύρες επικοινωνίας, οι οποίες είναι κανάλια μεταβλητών που τους έχει ανατεθεί να λαμβάνουν μηνύματα για μια συγκεκριμένη διεργασία (Atlas και Seitz [1988]).

1.3 Υπολογισμός πλέγματος (Grid Computing)

Ο υπολογισμός πλέγματος (grid computing) αποτελεί σημαντική εξέλιξη των κατανεμημένων συστημάτων. Προσφέρει τεράστια υπολογιστική ισχύ, κατά την οποία επιτυγχάνεται συνεργασία πολλών υπολογιστών που βρίσκονται σε διαφορετικά σημεία του κόσμου. Ο κατανεμημένος υπολογισμός χρησιμοποιεί ένα παρόμοιο μοντέλο για να χωρίσει τους πόρους που είναι διαθέσιμοι σε ξεχωριστούς υπολογιστές. Στη συνέχεια, ο υπολογισμός πλέγματος ανεβάζει αυτούς τους ξεχωριστούς υπολογιστές σε ένα επόμενο επίπεδο συνδέοντας πολλούς υπολογιστές που βρίσκονται σε γεωγραφικά απομακρυσμένες περιοχές. Επιτυγχάνεται έτσι μια εκτεταμένη συνεργασία μεταξύ των υπολογιστών και βελτιστοποίηση της ανταλλαγής πόρων.

Ορισμένα οφέλη από τον υπολογισμό πλέγματος είναι(R.Buyya και M.Murshed [2002]):

- Ο υπολογισμός πλέγματος επιτρέπει σε οργανισμούς να χρησιμοποιήσουν πόρους από άλλα υπολογιστικά συστήματα ανεξάρτητα από τις διαφορετικές γεωγραφικά περιοχές που βρίσκονται. Ακόμα περιορίζει καταστάσεις, σύμφωνα με τις οποίες μερικοί υπολογιστές είναι δυνατόν να χρησιμοποιούνται εντατικά, ενώ άλλοι είναι διαθέσιμοι και δεν χρησιμοποιούνται (με τη χρήση του σωστού χρονοπρογραμματισμού).
- Οι οργανισμοί, χρησιμοποιώντας τον διαμοιρασμό αυτών των πόρων, μπορούν να αναβαθμίσουν δραματικά την ποιότητα και ταχύτητα των προϊόντων και των υπηρεσιών που διατίθενται να προσφέρουν, ταυτόχρονα με τη μείωση του κόστους τους.
- Επιτρέπει σε εταιρίες να έχουν πρόσβαση και να μοιραστούν απομακρυσμένες βάσεις δεδομένων. Αυτό είναι ιδιαίτερα ωφέλιμο στις ανθρωπιστικές επιστήμες και στις ερευνητικές κοινότητες όπου τεράστιος όγκος δεδομένων δημιουργούνται και αναλύονται κατά τη διάρκεια μιας ημέρας.

• Επιτρέπει σε οργανισμούς να συνεργαστούν μεταξύ τους για τη δημιουργία έργων, εφόσον μπορούν να διαμοιραστούν εφαρμογές, δεδομένα κτλ.
• Μπορεί να δημιουργήσει μια πολύ καλύτερη υπολογιστική υποδομή η οποία θα μπορεί πολύ εύκολα να ανταποκριθεί σε μεγάλες ή μικρές καταστροφές.
• Μπορεί να εκμεταλλευτεί υπολογιστική ισχύ υπολογιστών που βρίσκονται σε διαφορετικές χώρες, η οποία δεν χρησιμοποιείται. Για παράδειγμα, υπολογιστές οι οποίοι δεν χρησιμοποιούνται τη νύχτα στο Τόκιο, θα μπορούν να χρησιμοποιηθούν την ημέρα στην Βόρεια Αμερική.

1.4 Πρωτόκολλα επικοινωνίας MPI (Message Passing Interface) και PVM (Parallel Virtual Machine)

1.4.1 Παράλληλα περιβάλλοντα (Parallel Environments)

Τα παράλληλα περιβάλλοντα προσφέρουν τη δυνατότητα στα μεμονωμένα προγράμματα να διανεμηθεί ο φόρτος εργασίας τους σε διάφορους επεξεργαστές, οι οποίοι αναλαμβάνουν τον παράλληλο υπολογισμό, με συνέπεια τη μείωση του χρόνου εκτέλεσης κάθε κώδικα. Το MPI (Message Passing Interface) και το PVM (Parallel Virtual Machine) προσφέρουν Διάμεσα Προγραμματιστικών Εφαρμογών (Application Programmer Interfaces, API) που επιτρέπουν στον υπολογισμό να διανεμηθεί στους πολλαπλούς επεξεργαστές σε διαφορετικές μηχανές.

1.4.2 Παράλληλη εικονική μηχανή (Parallel Virtual Machine, PVM)

Η Παράλληλη Εικονική Μηχανή (Parallel Virtual Machine) είναι ένα πακέτο λογισμικού (φορητό σύστημα προγραμματισμού ανταλλαγής μηνυμάτων), που επιτρέπει σε μια ετερογενή συλλογή υπολογιστών Unix ή/και Windows που συνδέονται μαζί σε ένα δίκτυο, να χρησιμοποιηθεί ως ένας ενιαίος, εύχρηστος πόρος υπολογισμού, μια "εικονική μηχανή". Το PVM αποτελεί μία βιβλιοθήκη προγραμματισμού για εφαρμογές ανταλλαγής μηνυμάτων σε μηχανές με κατανεμημένη μνήμη.

Η εικονική μηχανή μπορεί να αποτελείται από πλήθος ποικίλων τύπων, σε φυσικά μακρινές θέσεις. Οι εφαρμογές PVM μπορούν να αποτελούνται από οποιοδήποτε αριθμό ξεχωριστών διαδικασιών, ή συστατικά, που γράφονται σε C, C ++ και FORTRAN. Το σύστημα είναι φορητό σε μια ευρεία ποικιλία αρχιτεκτονικών, συμπεριλαμβανομένων των τερματικών σταθμών, των πολυεπεξεργαστών, των υπερυπολογιστών και PCs.

Η εικονική παράλληλη μηχανή επιτρέπει στους χρήστες να εκμεταλλευτούν το υπάρχον υλικό των υπολογιστών τους για να λύσουν μεγαλύτερα προβλήματα με μικρό συμπληρωματικό κόστος.

Επιπρόσθετα, το PVM είναι ένα ελεύθερο λογισμικό.

1.4.3 MPI (Message Passing Interface)

1.4.3.1 Τι είναι το MPI

Το MPI (Message Passing Interface) είναι μια βιβλιοθήκη λειτουργιών και μακροεντολών του υπολογιστή. Θεωρείται ότι είναι ένα από τα πρώτα πρότυπα που χρησιμοποιείται για παράλληλο προγραμματισμό. Το πρότυπο MPI χρησιμοποιείται σε προγράμματα C, C++ και FORTRAN, τα οποία προωθούν την ύπαρξη πολλαπλών επεξεργαστών με τη μεταφορά μηνυμάτων. Το MPI έχει εκτοπίσει τα περισσότερα εναλλακτικά συστήματα ανταλλαγής μηνυμάτων. Περιλαμβάνει ένα μεγάλο αριθμό συναρτήσεων οι οποίες διαχειρίζονται την επικοινωνία μεταξύ των επεξεργαστών.

Το MPI συντονίζει ένα πρόγραμμα στο οποίο τρέχουν πολλαπλές διαδικασίες παράλληλα σε ένα κατανεμημένο περιβάλλον μνήμης, μπορεί να χρησιμοποιηθεί όμως και σε ένα διαμοιραζόμενο σύστημα μνήμης. Κάθε διεργασία εκτελεί ένα τμήμα του ίδιου προγράμματος και χρησιμοποιεί τα δικά της δεδομένα για να κάνει τους υπολογισμούς της. Ο αριθμός των διεργασιών που δημιουργούνται σε ένα πρόγραμμα MPI είναι σταθερός και καθορίζεται από τον προγραμματιστή. Το MPI χρησιμοποιεί τις διαθέσιμες υπηρεσίες λειτουργικών συστημάτων για να δημιουργήσει παράλληλες διεργασίες με σκοπό την ανταλλαγή πληροφοριών μεταξύ αυτών των διεργασιών με τη διαβίβαση μηνυμάτων. Στις μηχανές εγκαθίσταται η βιβλιοθήκη του MPI και επομένως οι προγραμματιστές μπορούν να γράψουν κώδικες που περιέχουν κλήσεις συναρτήσεων του MPI (Message Passing Interface)

Επίσης το MPI παρέχει υποστήριξη για επικοινωνία κόμβου με κόμβο (point-to-point), συλλογική επικοινωνία, οργάνωση των διεργασιών σε ομάδες (grouping), οργάνωση των επικοινωνιών σε "κοινωνούς" (communicators), τοπολογίες επικοινωνίας διεργασιών καθώς και δυνατότητα διασύνδεσης με C και FORTRAN.

1.4.3.2 Λόγοι χρησιμοποίησης του MPI

✓ Τυποποίηση – το MPI είναι η μόνη βιβλιοθήκη διαβίβασης μηνυμάτων που μπορεί με τα σημερινά δεδομένα να θεωρηθεί ως πρότυπο.
✓ Φορητότητα – ο πηγαίος κώδικας δεν χρειάζεται τροποποίηση όταν ο προγραμματιστής εισάγει μια αίτηση σε μια διαφορετική πλατφόρμα η οποία υποστηρίζεται από το MPI.
✓ Απόδοση - οι εφαρμογές των προμηθευτών έχουν τη δυνατότητα να εκμεταλλεύονται εγγενή χαρακτηριστικά γνωρίσματα του υλικού για τη βελτιστοποίηση της απόδοσης.
✓ Λειτουργικότητα - (χρησιμοποιούνται πάνω από 115 ρουτίνες).
✓ Διαθεσιμότητα – ποικίλες εφαρμογές είναι διαθέσιμες, τόσο για τους προμηθευτές όσο και για τους δημόσιους τομείς.

1.4.3.3 Γενικά χαρακτηριστικά του MPI

Βασικά χαρακτηριστικά του MPI είναι τα ακόλουθα (Maui High Performance Computing Center (MHPCC), 1999):

Κατανεμημένη μνήμη (Distributed Memory)

Κάθε επεξεργαστής έχει την τοπική μνήμη του που μπορεί να προσπελαστεί άμεσα μόνο από την κεντρική μονάδα επεξεργασίας. Οι επεξεργαστές είναι συνδεδεμένοι σε δίκτυο και έτσι είναι εφικτή η μεταφορά των στοιχείων. Ακόμα δεν μπορούν να έχουν πρόσβαση άμεσα στον ίδιο πόρο μνήμης.

Διαβίβαση μηνυμάτων (Message Passing)

Κατά την διαβίβαση μηνυμάτων, αντιγράφονται τα δεδομένα από τη μνήμη κάποιου επεξεργαστή στη μνήμη ενός άλλου επεξεργαστή, εφόσον οι επεξεργαστές είναι δικτυακά συνδεδεμένοι. Στα συστήματα κατανεμημένης μνήμης, τα δεδομένα στέλνονται γενικά ως πακέτα πληροφοριών. Επίσης ένα μήνυμα αποτελείται από ένα ή περισσότερα πακέτα και συνήθως περιλαμβάνει ρουτίνες ή άλλες πληροφορίες ελέγχου.

Διεργασία (Process)

Η διεργασία είναι ένα σύνολο εκτελέσιμων οδηγιών (πρόγραμμα) που τρέχει σε έναν επεξεργαστή. Σε έναν επεξεργαστή μπορούν να εκτελεστούν μια ή περισσότερες διεργασίες. Κατά την διαβίβαση μηνυμάτων στο σύστημα, όλες οι διεργασίες επικοινωνούν μεταξύ τους καθώς και με την αποστολή των μηνυμάτων. Για λόγους αποδοτικότητας, τα συστήματα διαβίβασης μηνυμάτων συνδέουν γενικά μόνο μια διεργασία ανά επεξεργαστή.

Βιβλιοθήκη διαβίβασης μηνυμάτων (Message Passing Library)

Η βιβλιοθήκη διαβίβασης μηνυμάτων αποτελεί μια συλλογή συναρτήσεων που διασυνδέονται μέσα σε έναν κώδικα εφαρμογής που εκτελεί την αποστολή, την λήψη καθώς επίσης και άλλες λειτουργίες διαβίβασης μηνυμάτων.

Αποστολή / Λήψη (Send/Receive)

Η διαβίβαση μηνυμάτων περιλαμβάνει τη μεταφορά δεδομένων από μια διεργασία (αποστολή) σε μια άλλη διεργασία (λήψη) και απαιτεί τη συνεργασία και των δύο διεργασιών. Η διεργασία αποστολής διευκρινίζει τη θέση των στοιχείων, το μέγεθος, τον τύπο και τον προορισμό. Για κάθε διεργασία αποστολής πρέπει να έχει προβλεφθεί στο πρόγραμμα μια αντίστοιχη διαδικασία λήψης.

Σύγχρονος / Ασύγχρονος (Synchronous/Asynchronous)

Μία σύγχρονη λειτουργία αποστολής ολοκληρώνεται μόνο μετά από την αναγνώριση από την αντίστοιχη διεργασία λήψης ότι το μήνυμα παραλήφθηκε με ασφάλεια.

Μία ασύγχρονη λειτουργία αποστολής ολοκληρώνεται ακόμα κι αν η αντίστοιχη διεργασία λήψης δεν έχει λάβει πραγματικά το μήνυμα.

Προσωρινή αποθήκευση εφαρμογής (Απομονωτής εφαρμογής)(Application Buffer)

Αποτελεί την διεύθυνση μνήμης στην οποία φυλάσσεται το στοιχείο που πρόκειται να σταλεί ή να παραληφθεί. Χρησιμοποιείται μια μεταβλητή, η "inmsg". Ο απομονωτής εφαρμογής για την inmsg είναι η θέση μνήμης του προγράμματος όπου ανήκει η τιμή inmsg.

Εμποδιζόμενη Επικοινωνία (Blocking Communication)

Είναι δυνατόν μια ρουτίνα επικοινωνίας να εμποδιστεί όταν η ολοκλήρωση της κλήσης εξαρτάται από ορισμένα "γεγονότα" (εμποδιζόμενη επικοινωνία). Όσον αφορά την αποστολή, τα δεδομένα πρέπει να σταλούν επιτυχώς ή να αντιγραφούν με ασφάλεια στον απομονωτή

συστήματος έτσι ώστε ο απομονωτής εφαρμογής που περιείχε τα στοιχεία να είναι διαθέσιμος για επαναχρησιμοποίηση. Τέλος για τη λήψη, τα δεδομένα πρέπει να αποθηκευτούν με ασφάλεια στον απομονωτή λήψης έτσι ώστε να είναι έτοιμα για τη χρήση.

Μη-εμποδιζόμενη Επικοινωνία (Non-blocking Communication)

Μια ρουτίνα επικοινωνίας θεωρείται μη-εμποδιζόμενη όταν η ολοκλήρωση της κλήσης επιτυγχάνεται χωρίς αναμονή. Αυτό σημαίνει ότι οι ρουτίνες αποστολής και λήψης δεν περιμένουν καμία επικοινωνία να ολοκληρωθεί από την πλευρά του παραλήπτη, όπως η αντιγραφή μηνύματος από την τοπική μνήμη των χρηστών στη κοινή μνήμη των συστημάτων καθώς και η άφιξη του μηνύματος. Δεν χρησιμοποιείται ο απομονωτής εφαρμογής μετά από ολοκλήρωση της αποστολής non-blocking. Ο προγραμματιστής πρέπει να εξασφαλίσει ότι ο απομονωτής εφαρμογής είναι ελεύθερος για επαναχρησιμοποίηση.

1.4.3.4 Βασικές έννοιες ενός συστήματος MPI

Στην παρούσα παράγραφο θα παραθέσουμε ορισμένες βασικές έννοιες ενός συστήματος MPI (Maui High Performance Computing Center (MHPCC), 1999).

❖ *Πληροφοριοδότης (Communicator):* καλείται το αντικείμενο που χρησιμοποιείται από το MPI, το οποίο καθορίζει ποιες διεργασίες μπορούν να επικοινωνήσουν η μία με την άλλη.

❖ *MPI_COMM_WORLD:* ορίζεται ως ο αρχικός πληροφοριοδότης που περιλαμβάνει όλες τις διεργασίες σε μια ομάδα (σε μια παράλληλη εφαρμογή).

❖ *Τάξη (Rank):* είναι ο βαθμός μιας διεργασίας σε μια ρουτίνα επικοινωνίας, δηλαδή η αναφερόμενη διεργασία έχει αυτόν τον βαθμό μέσα στον συγκεκριμένο πληροφοριοδότη. Ο βαθμός αυτός είναι μοναδικός ακέραιος αριθμός που ορίζεται από το σύστημα όταν καλείται μια διεργασία, είναι συνεχόμενος και αρχίζει από το μηδέν. Συγκεκριμένα, με την τάξη (rank) προσδιορίζεται η πηγή και ο προορισμός ενός μηνύματος (if rank=0 do this / if rank=1 do that).

1.4.3.5 Βασικές εντολές του MPI

Το MPI χρησιμοποιεί σταθερό σχήμα για την ονομασία των ρουτινών του. Όλες οι ρουτίνες του MPI αρχίζουν με το πρόθεμα MPI_ και τον επόμενο πρώτο χαρακτήρα κεφαλαίο γράμμα. Οι υπόλοιποι χαρακτήρες των περισσότερων τύπων δεδομένων του MPI γράφονται με κεφαλαία γράμματα.

Περιγράφουμε τώρα ορισμένες από τις πιο βασικές εντολές που χρησιμοποιούνται στο πρωτόκολλο MPI (αναγράφονται οι κλήσεις των εντολών στη γλώσσα C):

➢ *MPI_Init*: Με τη λειτουργία αυτή αρχικοποιείται το περιβάλλον εκτέλεσης του MPI. Καλείται σε κάθε πρόγραμμα MPI, μόνο μία φορά και πριν από οποιεσδήποτε άλλες λειτουργίες MPI. Η κλήση της ρουτίνας MPI_Init περισσότερο από μια φορά κατά την διάρκεια εκτέλεσης ενός προγράμματος καταλήγει σε σφάλμα.

Δηλώνεται : MPI_Init (*argc, *argv)
MPI_INIT (ierr)

- *MPI_Comm_size* : Καθορίζει τον αριθμό των διαδικασιών στην ομάδα που συνδέεται με τον πληροφοριοδότη. Το MPI_COMM_WORLD καθορίζει τον αριθμό της χρήσης των διαδικασιών από μια αίτηση.

Δηλώνεται : MPI_Comm_size (comm,*size)
MPI_COMM_SIZE (comm,size,ierr)

- *MPI_Comm_rank*: Καθορίζει τον βαθμό της καλούμενης διαδικασίας μέσα στον πληροφοριοδότη. Αρχικά, σε κάθε διαδικασία ορίζεται μια μοναδική τάξη ακέραιων αριθμών μεταξύ 0 και τον αριθμό επεξεργαστών - 1 με τον πληροφοριοδότη MPI_COMM_WORLD. Ο βαθμός αυτός αναφέρεται ως ταυτότητα ID. Μια διαδικασία έχει έναν μοναδικό βαθμό μέσα σε έναν πληροφοριοδότη όταν συνδέεται με αυτόν.

Δηλώνεται : MPI_Comm_rank (comm,*rank)
MPI_COMM_RANK (comm,rank,ierr)

- *MPI_Abort*: Με τη λειτουργία αυτή ολοκληρώνονται όλες οι διαδικασίες MPI που συνδέονται με τον πληροφοριοδότη.

Δηλώνεται : MPI_Abort (comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)

- *MPI_Get_processor_name*: Η λειτουργία αυτή παίρνει το όνομα του επεξεργαστή στο οποίο εκτελείται η εντολή και επιστρέφει το μήκος του ονόματος. Ο απομονωτής για "το όνομα" πρέπει να είναι τουλάχιστον χαρακτήρες μεγέθους MPI_MAX_PROCESSOR_NAME.

Δηλώνεται:MPI_Get_processor_name
(*name,*resultlength)
MPI_GET_PROCESSOR_NAME
(name,resultlength,ierr)

- *MPI_Initialized:* Η λειτουργία αυτή δείχνει αν η λειτουργία MPI_Init έχει κληθεί. Επιστρέφει την τιμή (1) αληθές ή (0) ψευδές. Το MPI_Init καλείται μία φορά σε κάθε διαδικασία.

Δηλώνεται : MPI_Initialized (*flag)
MPI_INITIALIZED (flag,ierr)

- *MPI_Wtime*: Η λειτουργία αυτή επιστρέφει το χρόνο που έχει διανυθεί σε δευτερόλεπτα (διπλή ακρίβεια) στον καλούμενο επεξεργαστή.

Δηλώνεται : MPI_Wtime ()
MPI_WTIME ()

- *MPI_Wtick*: Η λειτουργία αυτή επιστρέφει τη λύση της λειτουργίας MPI_Wtime σε δευτερόλεπτα (διπλή ακρίβεια).

Δηλώνεται : MPI_Wtick ()
MPI_WTICK ()

- *MPI_Finalize*: Η λειτουργία αυτή ολοκληρώνει το περιβάλλον εκτέλεσης του MPI. Είναι η τελευταία ρουτίνα MPI που καλείται σε κάθε πρόγραμμα MPI.

Δηλώνεται : MPI_Finalize ()
MPI_FINALIZE (ierr).

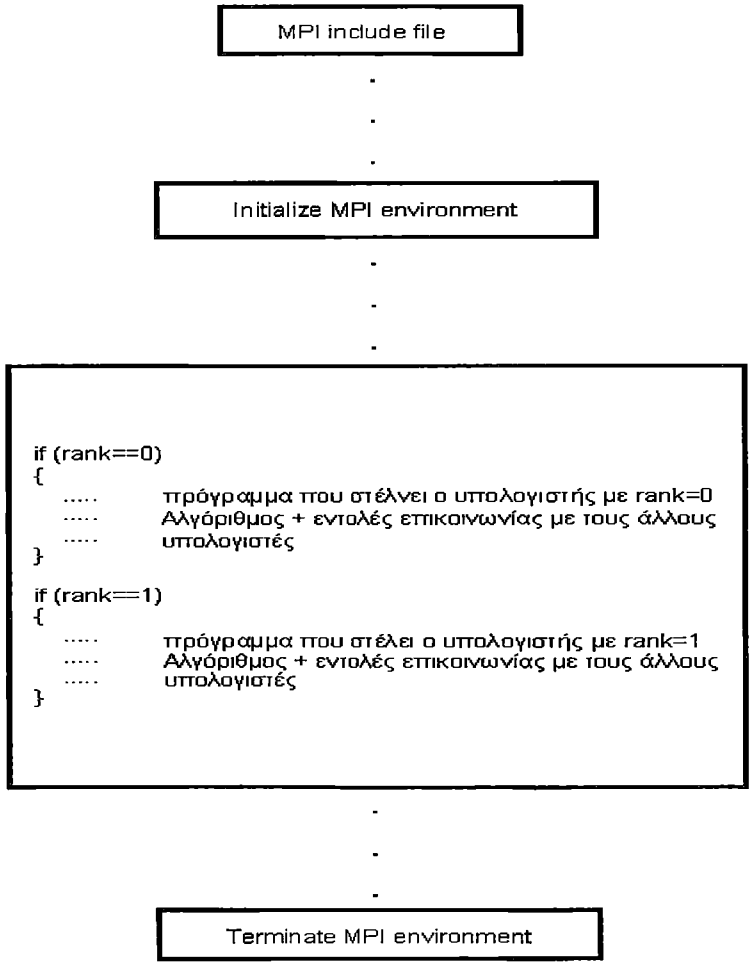
1.5 Ένα παράδειγμα κώδικα υλοποίησης εφαρμογής με το πρωτόκολλο MPI – Πρόγραμμα υπολογισμού αθροίσματος μονοδιάστατων πινάκων (athroisma.cpp)

Στην παρούσα παράγραφο θα αναλύσουμε ένα στοιχειώδες παράδειγμα που δίνει τα βασικά χαρακτηριστικά χρήσης του MPI. Το παρακάτω πρόγραμμα (athroisma.cpp) υπολογίζει το άθροισμα δυο μονοδιάστατων πινάκων, a[ ] και b[ ], σε παράλληλη μηχανή με χρήση του πρωτοκόλλου MPI.

```
#include "../mpich2-install/include/mpi.h"
#include <stdio.h>
#define maxrows 1000000

int main(int argc,char *argv[])
{
    int numtasks,rank,rc,sendtag=2001,retrtag=2002,i,
        avg_sum_per_processes,num_sum,num_sum_to_send,
        num_sum_to_receive,start,end;
    float a[maxrows],b[maxrows],sum1,sumfin;

    MPI_Status Stat;
    rc=MPI_Init(&argc,&argv);
    if(rc!=0)
    {
        printf("Error starting MPI program. Terminating. \n");
        MPI_Abort(MPI_COMM_WORLD,rc);
    }
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank==0)
    {
        for(i=0;i<=maxrows-1;i++)
        {
            a[i]=(float)i;
            b[i]=0.0;
        }
        num_sum=maxrows;
        avg_sum_per_processes=num_sum/numtasks;
        for(i=1;i<=numtasks-1;i++)
        {
            start=(i*avg_sum_per_processes);
            end=start+avg_sum_per_processes-1;
            if(i==(numtasks-1))
            {
                end=num_sum-1;
            }
            num_sum_to_send=end-start+1;
            rc=MPI_Send(&num_sum_to_send,1,MPI_INT,i,
                sendtag,MPI_COMM_WORLD);
            rc=MPI_Send(&a[start],num_sum_to_send,MPI_FLOAT,
                i,sendtag,MPI_COMM_WORLD);
        }
        sumfin=0.0;
        for(i=0; i<=avg_sum_per_processes-1;i++)
        {
            sumfin=sumfin+a[i];
        }
        printf("Sum from master = %f\n",sumfin);
        for(i=1;i<=numtasks-1;i++)
        {
            rc=MPI_Recv(&sum1,1,MPI_FLOAT,MPI_ANY_SOURCE,
                MPI_ANY_TAG,MPI_COMM_WORLD,&Stat);
            printf("partial sum from slave %d = %f\n",i,sum1);
            sumfin=sumfin+sum1;
        }
        printf("total sum = %f\n",sumfin);
```

```
    }
    else
    {
        rc=MPI_Recv(&num_sum_to_receive,1,MPI_INT,0,
            MPI_ANY_TAG,MPI_COMM_WORLD,&Stat);
        rc=MPI_Recv(&b,num_sum_to_receive,MPI_FLOAT,0,
            MPI_ANY_TAG,MPI_COMM_WORLD,&Stat);
        sum1=0;
        for(i=0;i<=num_sum_to_receive-1;i++)
        {
            sum1+=b[i];
        }
        rc=MPI_Send(&sum1,1,MPI_FLOAT,0,retrtag,MPI_COMM_WORLD);
    }
    MPI_Finalize();

}
```

Όπου: `numtasks` = αριθμός επεξεργαστών, `rank` = βαθμός διεργασίας, `sendtag` = αποστελλόμενη διεργασία, `retrtag` = λαμβανόμενη διεργασία, `num_sum` = σύνολο αριθμών (άθροισμα), `sum1` = αρχικό άθροισμα, `sumfin` = τελικό άθροισμα, `avg_sum_per_processes` = μέσος όρος αθροίσματος ανά διεργασία, `num_sum_to_send` = σύνολο αριθμών που αποστέλλεται, `num_sum_to_receive` = σύνολο αριθμών που λαμβάνεται.
Ο σκελετός ενός προγράμματος MPI απεικονίζεται στο ακόλουθο Σχήμα 1.5:



Σχήμα 1.5 – Απεικόνιση παράλληλου προγράμματος

Αρχικά ορίζεται η βιβλιοθήκη που χρησιμοποιείται για τη δημιουργία του προγράμματος. Η οδηγία #include οδηγεί τον μεταγλωττιστή της C να προσθέσει τα συστατικά ενός αρχείου συμπερίληψης στο συγκεκριμένο πρόγραμμα κατά την μεταγλώττιση.

```
#include "../mpich2-install/include/mpi.h"
#include <stdio.h>
```

Καθορισμός των στοιχείων των πινάκων (=1000000), χρησιμοποιώντας την εντολή define

```
#define maxrows 1000000
```

Αρχή κυρίως προγράμματος με την εντολή int main( )

```
int main(int argc,char *argv[])
```

Δήλωση των μεταβλητών που χρησιμοποιούνται στο πρόγραμμα

```
int numtasks,rank,rc,sendtag=2001,retrtag=2002,i,
    avg_sum_per_processes,num_sum,num_sum_to_send,
    num_sum_to_receive,start,end;
float a[maxrows],b[maxrows],sum1,sumfin;
```

Στο σημείο αυτό πρέπει να αναφερθεί ότι γίνεται αντιγραφή του αρχικού προγράμματος σε n επεξεργαστές παράλληλα συνδεδεμένους. Αρχικοποιείται το περιβάλλον εκτέλεσης του MPI, όπου ενεργοποιείται ένας δακτύλιος, κατά τον οποίο τα ορίσματα της γραμμής εντολών του MPI εκκινούν το περιβάλλον εκτέλεσής του σε όλες τις παράλληλες μηχανές. Η ρουτίνα MPI_Init καλείται μία φορά και πριν από οποιαδήποτε κλήση ρουτινών MPI· η κλήση της ρουτίνας παραπάνω από μια φορά κατά τη διάρκεια εκτέλεσης ενός προγράμματος καταλήγει σε σφάλμα.

```
MPI_Status Stat;
rc=MPI_Init(&argc,&argv);
```

Στη συνέχεια με μια δήλωση if τυπώνεται ένα μήνυμα λάθους αρχικοποίησης του προγράμματος αν rc!=0 και ολοκληρώνονται όλες οι διαδικασίες του MPI που συνδέονται με τον πληροφοριοδότη. Καθορίζεται ο αριθμός των διαδικασιών στην ομάδα που συνδέεται με τον πληροφοριοδότη καθώς επίσης και ο βαθμός της καλούμενης διαδικασίας.

```
if(rc!=0)
{
    printf("Error starting MPI program. Terminating. \n");
    MPI_Abort(MPI_COMM_WORLD,rc);
}

MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Αν ο αριθμός του μηχανήματος είναι 0 (rank==0) , δηλαδή είναι ο master , τότε στέλνει σε κάθε άλλο μηχάνημα τον ίδιο αριθμό στοιχείων του πίνακα για τον υπολογισμό του αθροίσματος για κάθε διαδικασία. Έπειτα υπολογίζει και στέλνει σε κάθε διαδικασία τα στοιχεία ώστε να επιτευχθεί ο υπολογισμός του αθροίσματος τους (num_sum_to_send).

```
if(rank==0)
{
    for(i=0;i<=maxrows-1;i++)
    {
        a[i]=(float)i;
        b[i]=0.0;
    }
    num_sum=maxrows;
    avg_sum_per_processes=num_sum/numtasks;
    for(i=1;i<=numtasks-1;i++)
    {
        start=(i*avg_sum_per_processes);
        end=start+avg_sum_per_processes-1;
        if(i==(numtasks-1))
        {
            end=num_sum-1;
        }
        num_sum_to_send=end-start+1;
        rc=MPI_Send(&num_sum_to_send,1,MPI_INT,i,
            sendtag,MPI_COMM_WORLD);
        rc=MPI_Send(&a[start],num_sum_to_send,MPI_FLOAT,
            i,sendtag,MPI_COMM_WORLD);
    }
```

Ο master υπολογίζει το άθροισμα των στοιχείων που αντιστοιχούν στη δική του καταχώρηση και τυπώνεται ανάλογο μήνυμα στην οθόνη

```
    sumfin=0.0;
    for(i=0; i<=avg_sum_per_processes-1;i++)
    {
        sumfin=sumfin+a[i];
    }
    printf("Sum from master = %f\n",sumfin);
```

Τέλος λαμβάνει τα αποτελέσματα των αθροισμάτων από τους slaves, υπολογίζει το τελικό άθροισμα και τυπώνεται μήνυμα στην οθόνη

```
    for(i=1;i<=numtasks-1;i++)
    {
        rc=MPI_Recv(&sum1,1,MPI_FLOAT,MPI_ANY_SOURCE,
            MPI_ANY_TAG,MPI_COMM_WORLD,&Stat);
        printf("partial sum from slave %d = %f\n",i,sum1);
        sumfin=sumfin+sum1;
    }
    printf("total sum = %f\n",sumfin);
}
```

Διαφορετικά, αν το node είναι ένας από τους slaves, δηλαδή αν rank != 0, τότε λαμβάνει από οποιοδήποτε tag τα στοιχεία του πίνακα, υπολογίζει το άθροισμα και στέλνει το αποτέλεσμα στον master

```
    else
    {
        rc=MPI_Recv(&num_sum_to_receive,1,MPI_INT,0,
            MPI_ANY_TAG,MPI_COMM_WORLD,&Stat);
        rc=MPI_Recv(&b,num_sum_to_receive,MPI_FLOAT,0,
            MPI_ANY_TAG,MPI_COMM_WORLD,&Stat);
        sum1=0;
        for(i=0;i<=num_sum_to_receive-1;i++)
        {
            sum1+=b[i];
        }
        rc=MPI_Send(&sum1,1,MPI_FLOAT,0,retrtag,MPI_COMM_WORLD);
    }
    MPI_Finalize();

}
```

2. ΠΑΡΑΛΛΗΛΟΙ ΑΛΓΟΡΙΘΜΟΙ ΕΠΙΛΥΣΗΣ ΓΡΑΜΜΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

2.1 Εισαγωγή

Στο παρόν κεφάλαιο θα αναλύσουμε δύο γνωστούς αλγόριθμους επίλυσης γραμμικών συστημάτων, τους αλγορίθμους Gauss και Jacobi, όπου συγκρίνονται η σειριακή και η παράλληλη υλοποίηση καθενός εκ των αλγορίθμων.

Σύστημα γραμμικών εξισώσεων ή *ανισώσεων* ή αλλιώς *γραμμικό σύστημα* είναι ένα σύνολο από γραμμικές εξισώσεις ή ανισώσεις με τους ίδιους αγνώστους, τους οποίους προσπαθούμε να προσδιορίσουμε ώστε να επαληθεύουν όλες τις εξισώσεις ή ανισώσεις του συνόλου.

Η πιο απλή μη τετριμμένη περίπτωση γραμμικού συστήματος είναι όταν έχουμε δυο άγνωστες μεταβλητές:

$$ax + by = c$$
$$a'x + b'y = c'$$

Για παράδειγμα, ένα σύστημα γραμμικών εξισώσεων δύο άγνωστων μεταβλητών είναι το:

$$3x + 5y = 2$$
$$2x - 7y = 9$$

ενώ αντίστοιχα σύστημα γραμμικών ανισώσεων είναι το:

$$3x + 5y \leq 2$$
$$2x - 7y > 9$$

Ένα σύστημα γραμμικών εξισώσεων μπορεί να είναι αδύνατο (καμία λύση), να έχει μοναδική λύση ή να είναι αόριστο (άπειρες λύσεις).

2.2 Αλγόριθμος Gauss

2.2.1 Μαθηματική περιγραφή

Υποθέτουμε ένα γραμμικό σύστημα της μορφής της εξίσωσης:

$$A * X = B \qquad (1.1)$$

όπου A είναι πίνακας $n*n$, και X, B είναι πίνακες $n*1$. Έστω το σύστημα έχει μοναδική λύση που δίνεται από την παρακάτω εξίσωση:

$$X = A^{-1} * B \tag{1.2}$$

Η εύρεση της λύσης με την μέθοδο των οριζουσών είναι πολύ χρονοβόρα (πολυπλοκότητας $O(N!)$) και άρα ασύμφορη, εκτός αν το σύστημα που θέλουμε να επιλύσουμε έχει μικρή διάσταση. Ο συνηθέστερος τρόπος επίλυσης συστημάτων της μορφής αυτής είναι μια παραλλαγή του αλγόριθμου Gauss. Ο βασικός αλγόριθμος περιγράφεται από τα εξής βήματα:

α) Ορίζουμε τον επαυξημένο πίνακα των συντελεστών και των σταθερών όρων

$$\left|\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array}\right| = \left|\begin{array}{ccccc} a_{11} & a_{12} & \dots & a_{1n} & a_{1n+1} \\ a_{21} & a_{22} & \dots & a_{2n} & a_{2n+1} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} & a_{nn+1} \end{array}\right|$$

όπου τα στοιχεία των σταθερών όρων b_1, b_2, ..., b_n έχουν μετονομαστεί σε a_{1n+1}, a_{2n+1}, ..., a_{nn+1}.

β) Ξεκινώντας με **οδηγό στοιχείο** (pivot element) το 1° στοιχείο της διαγωνίου (a_{11}), εκτελούμε μια πράξη μεταξύ της πρώτης γραμμής και κάθε μιας από τις επόμενες γραμμές κατά τέτοιο τρόπο ώστε το στοιχείο της κάθε γραμμής ακριβώς κάτω από το οδηγό στοιχείο να γίνεται ίσο με μηδέν. Έτσι, στη 2η γραμμή για να γίνει μηδέν το στοιχείο (a_{21}) θα πρέπει να πολλαπλασιαστεί η πρώτη γραμμή με το συντελεστή $c = -\frac{a_{21}}{a_{11}}$, και να προστεθεί στη δεύτερη. Αντίστοιχα για να μηδενιστεί ο όρος (a_{31}) πρέπει να πολλαπλασιαστεί η πρώτη γραμμή με το $c = -\frac{a_{31}}{a_{11}}$, να προστεθεί στη τρίτη και ούτω καθεξής έως ότου όλοι οι όροι κάτω από το οδηγό στοιχείο να γίνουν μηδέν. Μετά την εκτέλεση της πρώτης γραμμοπράξης ο πίνακας παίρνει τη μορφή:

$$\left|\begin{array}{ccccc} a_{11} & a_{12} & \dots & a_{1n} & a_{1n+1} \\ 0 & a'_{22} & \dots & a'_{2n} & a'_{2n+1} \\ 0 & a'_{32} & \dots & \dots & \dots \\ 0 & a'_{42} & \dots & a'_{nn} & a'_{nn+1} \end{array}\right|$$

Στα επόμενα, για συντομία στο συμβολισμό παραλείπουμε τους τόνους από τα στοιχεία του πίνακα, εννοούμε ότι μετά από κάθε γραμμοπράξη τα στοιχεία της αντίστοιχης γραμμής αντικαθίστανται από νέες τιμές, συνεχίζοντας τη διαδικασία, θέτουμε ως οδηγό στοιχείο το δεύτερο στοιχείο της διαγωνίου (a_{22}). Ακολουθούμε την ίδια διαδικασία για να κάνουμε μηδενικούς όλους τους όρους κάτω από το στοιχείο (a_{22}). Όταν τελειώσουμε και με τη δεύτερη στήλη συνεχίζουμε, με τον ίδιο ακριβώς τρόπο μέχρι να φτάσουμε στο τελευταίο στοιχείο της διαγωνίου, το στοιχείο (a_{nn}), κάτω από το οποίο δεν υπάρχουν άλλοι όροι για να

μηδενίσουμε. Μετά την εκτέλεση όλων των πράξεων, ο επαυξημένος πίνακας αποκτά την ακόλουθη τριγωνική μορφή:

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} & a_{1n+1} \\ 0 & a_{22} & \dots & a_{2n} & a_{2n+1} \\ 0 & 0 & \dots & \dots & \dots \\ 0 & 0 & 0 & a_{nn} & a_{nn+1} \end{vmatrix}$$

όπου όλα τα στοιχεία κάτω από τη διαγώνιο είναι ίσα με μηδέν. Η κατάσταση αυτή του πίνακα επιτρέπει να υπολογισθούν απευθείας οι λύσεις με τη διαδικασία η οποία είναι γνώστη ως *πίσω αντικατάσταση (backward substitution)*. Έτσι, ξεκινώντας από την τελευταία γραμμή, στη οποία η εμφανιζόμενη μετασχηματισμένη εξίσωση θα έχει μόνο άγνωστο τον x_n, με συντελεστή (a_{nn}), βρίσκουμε κατευθείαν

$$x_n = \frac{1}{a_{nn}} a_{nn+1} \qquad (1.3)$$

με δεδομένη την τιμή του x_n, αντικαθιστούμε τώρα στην αμέσως προηγούμενη εξίσωση και βρίσκουμε την τιμή του x_{n-1}. Στη συνέχεια αντικαθιστούμε στην αμέσως προηγούμενη κ.ο.κ.. Έτσι βρίσκουμε διαδοχικά

$$\begin{aligned} x_{n-1} &= \frac{1}{a_{n-1n-1}} (a_{n-1n-1} - a_{n-1n} x_n) \\ x_{n-2} &= \frac{1}{a_{n-2n-2}} (a_{n-2n+1} - a_{n-2n} x_n - a_{n-2n-1} x_{n-1}) \end{aligned} \qquad (1.4)$$

και γενικά

$$x_i = \frac{1}{a_{ii}} (a_{in+1} - \sum_{k=i+1}^{n} a_{ik} x_k) \qquad (1.5)$$

Επομένως, κάθε νέος άγνωστος x_i υπολογίζεται από την σχέση της εξίσωσης (1.5) ως συνάρτηση των συντελεστών, του σταθερού όρου της αντίστοιχης εξίσωσης και των τιμών των μεταβλητών x_k με $k > i$, οι οποίες έχουν υπολογισθεί προηγουμένως. Εδώ έχουμε υποθέσει ότι το σύστημα έχει μοναδική λύση, οπότε κανένα από τα στοιχεία $a_{ii}, i = 1,\dots,n$ δεν μηδενίζεται.

Παράδειγμα:

Θα λύσουμε στη συνέχεια με τη μέθοδο Gauss το ακόλουθο σύστημα της εξίσωσης (1.6):

$$\begin{aligned} 5_{x1} + 2_{x2} + x_3 + 2_{x4} &= 20 \\ 3_{x1} + x_2 + 2_{x3} + 4_{x4} &= 27 \\ x_1 + x_2 + x_3 + x_4 &= 10 \\ -x_1 + x_2 - x_3 + x_4 &= 2 \end{aligned} \qquad (1.6)$$

Ο επαυξημένος πίνακας που προκύπτει από το σύστημα της εξίσωσης (1.6) είναι αυτός που φαίνεται στην εξίσωση (1.7):

$$\left|\begin{array}{ccccc} 5 & 2 & 1 & 2 & 20 \\ 3 & 1 & 2 & 4 & 27 \\ 1 & 1 & 1 & 1 & 10 \\ -1 & 1 & -1 & 1 & 2 \end{array}\right| \tag{1.7}$$

Αρχικά θα απαλείψουμε όλους τους όρους που βρίσκονται κάτω από τον όρο a_{11}. Στα αριστερά του πίνακα σημειώνουμε τον συντελεστή με τον οποίο θα πολλαπλασιάσουμε την πρώτη γραμμή ώστε όταν την πολλαπλασιάσουμε με αυτόν και την προσθέσουμε στην αντίστοιχη σειρά να μηδενιστεί ο συντελεστής που βρίσκεται κάτω από τον όρο της διαγωνίου. Έχουμε λοιπόν διαδοχικά τις ακόλουθες απαλοιφές

$$\begin{array}{c|ccccc} & 5 & 2 & 1 & 2 & 20 \\ -3/5 & 3 & 1 & 2 & 4 & 27 \\ -1/5 & 1 & 1 & 1 & 1 & 10 \\ 1/5 & -1 & 1 & -1 & -1 & 2 \end{array} \Rightarrow \left|\begin{array}{ccccc} 5 & 2 & 1 & 2 & 20 \\ 0 & -1/5 & 7/5 & 14/5 & 15 \\ 0 & 3/5 & 4/5 & 3/5 & 6 \\ 0 & 7/5 & -4/5 & 7/5 & 6 \end{array}\right| \tag{1.8}$$

στη συνέχεια μηδενίζουμε τους όρους κάτω από το δεύτερο στοιχείο της διαγωνίου, το στοιχείο (a_{22}). Έτσι έχουμε τις απαλοιφές:

$$\begin{array}{c|ccccc} & 5 & 2 & 1 & 2 & 20 \\ & 0 & -1/5 & 7/5 & 14/5 & 15 \\ 3 & 0 & 3/5 & 4/5 & 3/5 & 6 \\ 7 & 0 & 7/5 & -4/5 & 7/5 & 6 \end{array} \Rightarrow \left|\begin{array}{ccccc} 5 & 2 & 1 & 2 & 20 \\ 0 & -1/5 & 7/5 & 14/5 & 15 \\ 0 & 0 & 5 & 9 & 51 \\ 0 & 0 & 9 & 21 & 111 \end{array}\right| \tag{1.9}$$

και τέλος μηδενίζουμε τους όρους κάτω από το τρίτο στοιχείο της διαγωνίου, το στοιχείο (a_{33}). Έτσι έχουμε τις απαλοιφές:

$$\begin{array}{c|ccccc} & 5 & 2 & 1 & 2 & 20 \\ & 0 & -1/5 & 7/5 & 14/5 & 15 \\ & 0 & 0 & 5 & 9 & 51 \\ -9/5 & 0 & 0 & 0 & 24/5 & 96/5 \end{array} \Rightarrow \left|\begin{array}{ccccc} 5 & 2 & 1 & 2 & 20 \\ 0 & -1/5 & 7/5 & 14/5 & 15 \\ 0 & 0 & 5 & 9 & 51 \\ 0 & 0 & 0 & 24/5 & 96/5 \end{array}\right| \tag{1.10}$$

οπότε, μέσω της εξίσωσης (1.10) η οποία είναι η τελική κατάσταση του πίνακα της μεθόδου Gauss μπορούμε να υπολογίσουμε μία μία τις τιμές των αγνώστων ξεκινώντας από τον τελευταίο, δηλαδή τον x_4. Έτσι προκύπτει ότι:

$$x_4 = \frac{1}{\left(\frac{24}{5}\right)}\left(\frac{96}{5}\right) = 4 \tag{1.11}$$

Οι τιμές στην εξίσωση (1.11) έχουν μπει σε παρένθεση για να συμφωνούν ακριβώς με τον τρόπο με τον οποίο περιγράφονται στην εξίσωση (1.3). Στη συνέχεια αφού γνωρίζουμε την

τιμή του x_4 υπολογίζουμε τις τιμές των υπολοίπων αγνώστων σύμφωνα με την εξίσωση (1.5). Έτσι έχουμε:

$$x_3 = \frac{51-9x_4}{5} = \frac{51-9*4}{5} = 3 \tag{1.12}$$

$$x_2 = \frac{15-\frac{14}{5}x_4-\frac{7}{5}x_3}{-\frac{1}{5}} = 2 \tag{1.13}$$

και

$$x_1 = \frac{20-2x_4-x_3-2x_2}{5} = 1 \tag{1.14}$$

Οι τιμές αυτές που υπολογίσαμε αποτελούν και τη μοναδική λύση του συστήματος της εξίσωσης (1.6).

2.2.2 Πολυπλοκότητα του αλγορίθμου Gauss

Για το αλγόριθμο Gauss η πολυπλοκότητα υπολογίζεται ως εξής:

$$\left[\begin{array}{ccccc|c} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{array}\right]$$

Έστω ότι έχουμε τον παραπάνω πίνακα με $N*(N+1)$ στοιχεία. Τότε στο Ν-οστό βήμα του αλγόριθμου έχουμε :

1. $a[ij] = a[ij] - c[a_{1j}], c = \frac{a[i1]}{a[11]}$.
2. Για την εύρεση του πολλαπλασιαστικού παράγοντα κάθε γραμμοπράξης απαιτείται 1 διαίρεση μια και $c = \frac{a[ik]}{a[kk]}$,

   Για την εκτέλεση της γραμμοπράξης απαιτούνται $N+1-k$ πράξεις

   Οι γραμμοπράξεις είναι $N-k$.

Παραδείγματος χάριν, με $N=5$, οι πράξεις είναι συνοπτικά:

$4*(6+1)$

$3*(5+1)$

$2*(4+1)$

$1*(3+1)$

3. Γενικά, για το επαυξημένο πίνακα $N*(N+1)$ έχουμε:

$$(N-1)(N+2)+((N-1)-1)((N-1)+2)+((N-2)-1)((N-2)+2)+1(2+2)$$
$$=\frac{N(N+1)(2N-1)}{6}-1+\frac{N(N+1)}{2}-1-2(N-1)\approx\frac{2N^3}{6}+\frac{N^2}{2}-2N$$

Επομένως, ο αλγόριθμος Gauss έχει πολυπλοκότητα τάξης $O\left(\frac{N^3}{3}\right)$.

2.2.3 Σειριακός αλγόριθμος

Στο παρόν υποκεφάλαιο παραθέτουμε την επίλυση του γραμμικού συστήματος σειριακά:

```
#include <stdio.h>
#include <math.h>
#define n 1000

void main()
{
    int i,j,k,imax;
    float x[n],a[n][n+1],c,sum,max,sw,det;
    printf("dwse stoixeia pinaka: \n");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%f",&a[i][j]);
        for(i=0;i<n;i++)
        {
            max=float(fabs(a[i][i]));
            imax=i;
            for(j=i+1;j<n;j++)
                if(fabs(a[j][i])>max)
                {
                    max=float(fabs(a[j][i]));
                    imax=j;
                }
                for(k=0;k<n+1;k++)
                {
                    sw=a[i][k];
                    a[i][k]=a[imax][k];
                    a[imax][k]=sw;
                }
                for(j=i+1;j<n;j++)
                {
                    c=-a[j][i]/a[i][i];
                    for (k=i;k<n+1;k++)
                        a[j][k]=c*a[i][k]+a[j][k];
                }
        }
        det=1;
        for(i=0;i<n;i++)
            det=det*a[i][i];
        printf("det = %.2f\n",det);
        if(det==0)
            printf("underrmited/impossible \n");
        else
            for(i=n-1;i>=0;i--)
            {
                sum=0;
                for(k=i+1;k<n;k++)
                    sum=sum+a[i][k]*x[k];
                x[i]=(1/a[i][i])*(a[i][n+1]-sum);
                printf("x[i] = %.2f\n",x[i]);
            }
}
```

Ο παραπάνω κώδικας δίνει την σειριακή λύση ενός γραμμικού συστήματος με την μέθοδο του Gauss. Παρακάτω θα δούμε τα βασικότερα κομμάτια κώδικα:

```
for(i=0;i<n;i++)
{
    max=float(fabs(a[i][i]));
    imax=i;
    for(j=i+1;j<n;j++)
        if(fabs(a[j][i])>max)
        {
            max=float(fabs(a[j][i]));
            imax=j;
        }
        for(k=0;k<n+1;k++)
        {
            sw=a[i][k];
            a[i][k]=a[imax][k];
            a[imax][k]=sw;
        }
```

Σε αυτό το κομμάτι κώδικα γίνεται με τις κατάλληλες εντολές for η εύρεση του οδηγού στοιχείου στο πίνακα που του έχουμε δώσει, μετά ακολουθούν οι γραμμοπράξεις, απαλοιφή gauss, όπως φαίνεται στις παρακάτω γραμμές κώδικα:

```
for(j=i+1;j<n;j++)
{
    c=-a[j][i]/a[i][i];
    for (k=i;k<n+1;k++)
        a[j][k]=c*a[i][k]+a[j][k];
}
```

Παρακάτω γίνεται ο υπολογισμός της ορίζουσας του τριγωνικού πίνακα με τις ακόλουθες εντολές:

```
det=1;
for(i=0;i<n;i++)
    det=det*a[i][i];
printf("det = %.2f\n",det);
```

Τέλος, με τις παρακάτω εντολές πραγματοποιείται η πίσω αντικατάσταση με την οποία δίνεται η λύση του συστήματος Gauss, και τυπώνονται στην οθόνη τα αποτελέσματα των άγνωστων x_i:

```
if(det==0)
    printf("underrmited/impossible \n");
else
    for(i=n-1;i>=0;i--)
    {
        sum=0;
        for(k=i+1;k<n;k++)
            sum=sum+a[i][k]*x[k];
        x[i]=(1/a[i][i])*(a[i][n+1]-sum);
        printf("x[i] = %.2f\n",x[i]);
    }
```

2.2.4 Στρατηγική παραλληλοποίησης του γραμμικού συστήματος Gauss

Έστω ότι έχουμε διαθέσιμους 10 επεξεργαστές. Ο πρώτος επεξεργαστής ονομάζεται master και οι υπόλοιποι 9 slaves.

α) Ο master έχει στη διάθεση του ολόκληρο το σύστημα Gauss και καλείται να στείλει στους slaves την γραμμή που περιέχει κάθε φορά το οδηγό στοιχείο καθώς και ορισμένες από τις υπόλοιπες γραμμές του πίνακα, για να πραγματοποιηθεί από τους slaves η απαλοιφή Gauss.

Μια τέτοια κατανομή των γραμμών του πίνακα είναι δυνατή δεδομένου ότι η εκτέλεση των γραμμοπράξεων μεταξύ της γραμμής του οδηγού στοιχείου και κάποιας άλλης γραμμής του πίνακα είναι ανεξάρτητη από την εκτέλεση των γραμμοπράξεων της ίδιας γραμμής οδηγού στοιχείου με οποιαδήποτε άλλη γραμμή του πίνακα.

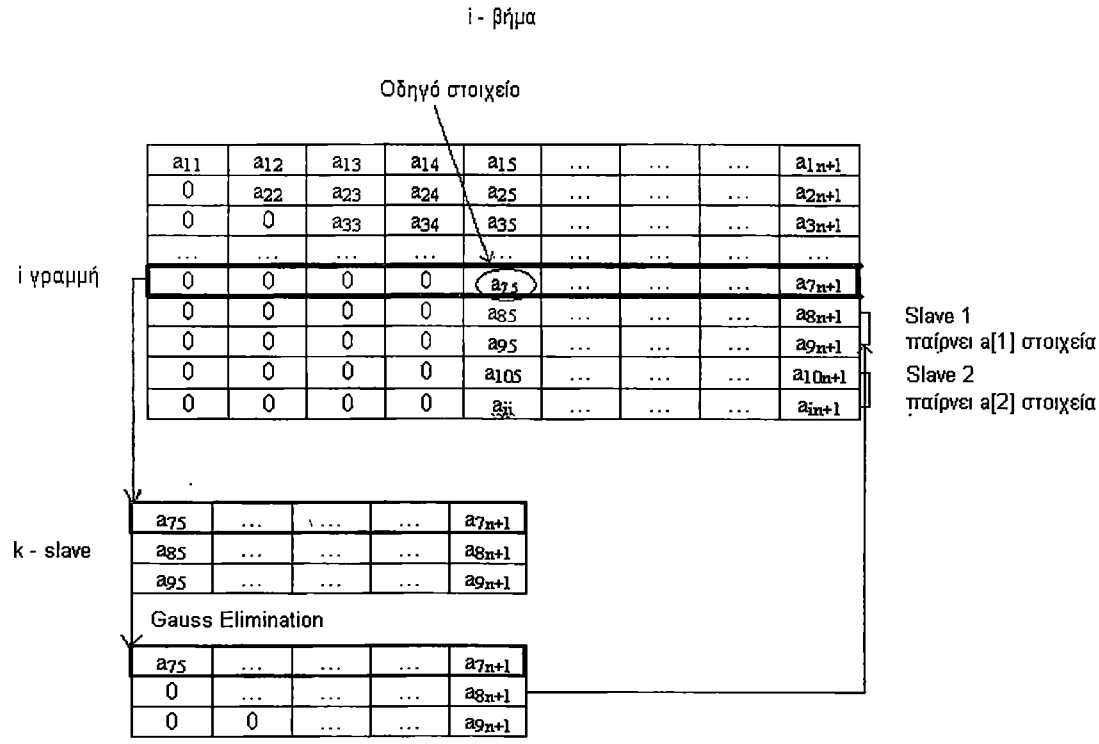
β) Πραγματοποιείται η απαλοιφή Gauss από τους slaves.

γ) Περαιτέρω, απαλοιφή Gauss πραγματοποιεί και ο master κρατώντας για τον εαυτό του κάποιες γραμμές.

δ) Ύστερα ο master λαμβάνει τις «απαντήσεις» από τους slaves και τελειώνει ο μεγάλος βρόγχος της απαλοιφής.

ε) Τέλος, ο master υπολογίζει την ορίζουσα, πραγματοποιεί την πίσω αντικατάσταση (***backward substitution***) και δίνει τη λύση του γραμμικού συστήματος Gauss.

Ας σημειωθεί ότι η πίσω αντικατάσταση είναι διαδικασία που απαιτεί συνολικά $O(N^2)$ πολλαπλασιασμούς, ενώ είναι αναγκαστικά διαδικασία σειριακή μιας και κάθε υπολογισμός απαιτεί τη γνώση του προηγούμενου στοιχείου x_i. Ωστόσο, το επιπλέον κόστος από την $O(N^2)$ διαδικασία είναι μικρό σε σχέση με το όφελος από την παραλληλοποίηση της απαλοιφής Gauss.



Σχήμα 2.1 – Στρατηγική Παραλληλοποίησης γραμμικού συστήματος Gauss

2.2.5 Παράλληλος αλγόριθμος – Περιγραφή

```
#include "../mpich2-install/include/mpi.h"
#include <stdio.h>
#include <math.h>

void moirasma(int ntotal,int nmachine,int a[]);

#define n 1000

int main(int argc,char *argv[])
{
    int i,j,k,imax,numtasks,rc,rank,mpi_any_tag,
        grammes_ana_process,tag,mpi_any_source,itask,
        pmachine[100],icount,ngrammwn,pmach2;
    float x[n],a[n][n+1],b[n][n+1],c,sum,max,sw,det;
    MPI_Status Stat;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

//------------------------------ MASTER ------------------
    if(rank==0)
    {
//.........gemisma pinaka
    printf("dwse stoixeia pinaka: \n");
        for(i=0;i<n;i++)
        {
            for(j=0;j<n+1;j++)
            {
                scanf("%f",&a[i][j]);
            }
        }

//##########BIG LOOP: GAUSSIAN ELIMINATION ##############################
            for(i=0;i<n;i++)
            {
                printf("loop odigou stoixeiou %d\n",i);

//............evresi odigou stoixeiou kai enallagi
                max=float(fabs(a[i][i]));
                imax=i;
                for(j=i+1;j<n;j++)
                {
                    if(fabs(a[j][i])>max)
                    {
                        max=float(fabs(a[j][i]));
                        imax=j;
                    }
                }
                for(k=0;k<n+1;k++)
                {
                    sw=a[i][k];
                    a[i][k]=a[imax][k];
                    a[imax][k]=sw;
                }
```

```
//.............apostoli grammon stous slaves
                ngrammwn=n-i-1;
                moirasma(ngrammwn,numtasks,pmachine);
                icount=pmachine[0];
                for(itask=1;itask<=numtasks-1;itask++)
                {
                    for(k=0;k<n+1;k++)
                    {
                        b[0][k]=a[i][k];
                    }
                    for(j=1;j<=pmachine[itask];j++)
                        for(k=0;k<n+1;k++)
                        {
                            b[j][k]=a[icount+i+j][k];
                        }
            rc=MPI_Send(&pmachine[itask],1,MPI_INT,
                itask,1000,MPI_COMM_WORLD);
            if(pmachine[itask]>0)
            {
                rc=MPI_Send(&b[0][0],(pmachine[itask]+1)*(n+1),
                    MPI_FLOAT,itask,1001,MPI_COMM_WORLD);
                icount+=pmachine[itask];
            }
    }
    printf("send to slaves accomplished\n");
//.............apaloifi gauss stis grammes tou master
                for(j=i+1;j<n;j++)
                {
                    c=-a[j][i]/a[i][i];
                    for(k=i;k<n+1;k++)
                        a[j][k]=c*a[i][k]+a[j][k];
                }
//.............lipsi grammon apo slaves meta tin elimination
                icount=pmachine[0];
                for(itask=1;itask<=numtasks-1;itask++)
                {
                    if(pmachine[itask]>0)
                    {
                        rc=MPI_Recv(&b[0][0],(pmachine[itask]+1)*(n+1),
                            MPI_FLOAT,itask,1002,MPI_COMM_WORLD,&Stat);
                    }
                    for(j=1;j<=pmachine[itask];j++)
                        for(k=0;k<n+1;k++)
                        {
                            a[icount+i+j][k]=b[j][k];
                        }
                        icount+=pmachine[itask];
                }
                printf("receiving from slaves accomplished\n");
            }
//###########END OF BIG LOOP: GAUSSIAN ELIMINATION##################

//..........ypologismos orizousas
            det=1;
            for(i=0;i<n;i++)
            {
                det=det*a[i][i];
            }
            printf("H orizousa einai:%.2f\n",det);
```

```
//.........piso antikatastasi
            if(det==0)
            {
                printf("H eksiswsh einai adunath: \n");
            }
            else
            {
                for(i=n-1;i>=0;i--)
                {
                    sum=0;
                    for(k=i+1;k<n;k++)
                    {
                        sum=sum+a[i][k]*x[k];
                    }
                    x[i]=(1/a[i][i])*(a[i][n]-sum);
                }
                printf("h lush einai :\n");
                for(i=0;i<n;i++)
                {
                    printf("x[%d]=%.2f\n",i,x[i]);
                }
            }
    }
// ---------------- END OF MASTER ----------------------------

// --------------- SLAVE -------------------------------------
    else
    {
        i=0;
        for(i=0;i<n;i++)
        {
            rc=MPI_Recv(&pmach2,1,MPI_INT,0,1000,
                MPI_COMM_WORLD,&Stat);
            if(pmach2>0)
            {
                rc=MPI_Recv(&b[0][0],(pmach2+1)*(n+1),MPI_FLOAT,
                    0,1001,MPI_COMM_WORLD,&Stat);
                for(j=1;j<=pmach2;j++)
                {
                    c=-b[j][i]/b[0][i];
                    for(k=i;k<n+1;k++)
                        b[j][k]=c*b[0][k]+b[j][k];
                }
                rc=MPI_Send(&b[0][0],(pmach2+1)*(n+1),MPI_FLOAT,
                    0,1002,MPI_COMM_WORLD);
            }
        }
    }
//--------------- SLAVE -------------------------------------------

    MPI_Finalize();
    return 0 ;
}

void moirasma(int ntotal,int nmachine,int a[])
{
    int i;
    int nperiseuma,ntask;
    ntask=ntotal/nmachine;
    for(i=0;i<=nmachine-1;i++)
    {
        a[i]=ntask;
    }
    nperiseuma=ntotal-ntask*nmachine;
    for(i=0;i<=nperiseuma-1;i++)
    {
        a[i]++;
    }
}
```

Σε αυτό το σημείο θα δούμε αναλυτικά την επίλυση του γραμμικού συστήματος Gauss

```
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Με τις παραπάνω γραμμές κώδικα γίνεται η έναρξη του δακτυλίου MPI. Καθορίζεται ο αριθμός των διαδικασιών στην ομάδα που συνδέεται με τον πληροφοριοδότη καθώς επίσης και ο βαθμός της καλούμενης διαδικασίας.

```
if(rank==0)
{
//.........gemisma pinaka
printf("dwse stoixeia pinaka: \n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<n+1;j++)
        {
            scanf("%f",&a[i][j]);
        }
    }
```

Αν ο βαθμός της διαδικασίας είναι 0, δηλαδή αν είναι ο master, γεμίζουμε τον πίνακα με την μέθοδο που μας δίνει την δυνατότητα να τρέχουμε κάθε φορά ένα τυχαίο γραμμικό σύστημα. Οι επόμενες γραμμές αναφέρονται στις εντολές που εκτελεί ο master.

```
//##########BIG LOOP: GAUSSIAN ELIMINATION ##############################
        for(i=0;i<n;i++)
        {
            printf("loop odigou stoixeiou %d\n",i);

//............evresi odigou stoixeiou kai enallagi
            max=float(fabs(a[i][i]));
            imax=i;
            for(j=i+1;j<n;j++)
            {
                if(fabs(a[j][i])>max)
                {
                    max=float(fabs(a[j][i]));
                    imax=j;
                }
            }
            for(k=0;k<n+1;k++)
            {
                sw=a[i][k];
                a[i][k]=a[imax][k];
                a[imax][k]=sw;
            }
```

Στις πρώτες γραμμές γίνεται η λεγόμενη "οδήγηση κατά γραμμή" (π.χ. Press et al., 1992). Μεταφέρουμε δηλαδή στην i-γραμμή εκείνη για την οποία το οδηγό στοιχείο είναι κατ' απόλυτη τιμή μεγαλύτερο από κάθε άλλο στοιχείο της ίδιας στήλης και γραμμής μεγαλύτερο του i. Αυτό γίνεται για να ελαχιστοποιούνται τα αριθμητικά σφάλματα κατά την εκτέλεση του αλγορίθμου.

```
//.............apostoli grammon stous slaves
                ngrammwn=n-i-1;
                moirasma(ngrammwn,numtasks,pmachine);
                icount=pmachine[0];
                for(itask=1;itask<=numtasks-1;itask++)
                {
                    for(k=0;k<n+1;k++)
                    {
                        b[0][k]=a[i][k];
                    }
                    for(j=1;j<=pmachine[itask];j++)
                        for(k=0;k<n+1;k++)
                        {
                            b[j][k]=a[icount+i+j][k];
                        }
                   rc=MPI_Send(&pmachine[itask],1,MPI_INT,
                       itask,1000,MPI_COMM_WORLD);
                   if(pmachine[itask]>0)
                   {
                       rc=MPI_Send(&b[0][0],(pmachine[itask]+1)*(n+1),
                           MPI_FLOAT,itask,1001,MPI_COMM_WORLD);
                       icount+=pmachine[itask];
                   }
            }
            printf("send to slaves accomplished\n");


//.............apaloifi gauss stis grammes tou master
                for(j=i+1;j<n;j++)
                {
                    c=-a[j][i]/a[i][i];
                    for(k=i;k<n+1;k++)
                        a[j][k]=c*a[i][k]+a[j][k];
                }
//.............lipsi grammon apo slaves meta tin elimination
                icount=pmachine[0];
                for(itask=1;itask<=numtasks-1;itask++)
                {
                    if(pmachine[itask]>0)
                   {
                       rc=MPI_Recv(&b[0][0],(pmachine[itask]+1)*(n+1),
                           MPI_FLOAT,itask,1002,MPI_COMM_WORLD,&Stat);
                   }
                   for(j=1;j<=pmachine[itask];j++)
                       for(k=0;k<n+1;k++)
                       {
                           a[icount+i+j][k]=b[j][k];
                       }
                        icount+=pmachine[itask];
                }
                printf("receiving from slaves accomplished\n");
            }
//##########END OF BIG LOOP: GAUSSIAN ELIMINATION#################
```

Πραγματοποιείται η αποστολή των γραμμών στους σκλάβους, αφού γίνει πρώτα έλεγχος μέσω της συνάρτησης `moirasma` για να είναι δίκαιη για όλους τους επεξεργαστές (βλ. παραπάνω). Η αποστολή γίνεται με τις κατάλληλες εντολές `MPI_Send`. Γίνεται απαλοιφή Gauss, μια διαδικασία που πραγματοποιείται ταυτόχρονα και στους σκλάβους. Πραγματοποιείται η λήψη των αποτελεσμάτων από την απαλοιφή Gauss που έγινε από τους σκλάβους, και εδώ τελειώνει ο μεγάλος βρόγχος της απαλοιφής.

```
//.........ypologismos orizousas
            det=1;
            for(i=0;i<n;i++)
            {
                det=det*a[i][i];
            }
            printf("H orizousa einai:%.2f\n",det);
```

Με τις παραπάνω εντολές γίνεται ο υπολογισμός της ορίζουσας από τον master.

```
//..........piso antikatastasi
            if(det==0)
            {
                printf("H eksiswsh einai adunath: \n");
            }
            else
            {
                for(i=n-1;i>=0;i--)
                {
                    sum=0;
                    for(k=i+1;k<n;k++)
                    {
                        sum=sum+a[i][k]*x[k];
                    }
                    x[i]=(1/a[i][i])*(a[i][n]-sum);
                }
                printf("h lush einai :\n");
                for(i=0;i<n;i++)
                {
                    printf("x[%d]=%.2f\n",i,x[i]);
                }
            }
    }
```

Τέλος, γίνεται η πίσω αντικατάσταση η οποία και δίνει τις λύσεις του συστήματος Gauss, εδώ τελειώνει και η εργασία του master.

```
// ---------------- SLAVE ----------------------------------------
    else
    {
        i=0;
        for(i=0;i<n;i++)
        {
            rc=MPI_Recv(&pmach2,1,MPI_INT,0,1000,
                MPI_COMM_WORLD,&Stat);
            if(pmach2>0)
            {
                rc=MPI_Recv(&b[0][0],(pmach2+1)*(n+1),MPI_FLOAT,
                    0,1001,MPI_COMM_WORLD,&Stat);
                for(j=1;j<=pmach2;j++)
                {
                    c=-b[j][i]/b[0][i];
                    for(k=i;k<n+1;k++)
                        b[j][k]=c*b[0][k]+b[j][k];
                }
                rc=MPI_Send(&b[0][0],(pmach2+1)*(n+1),MPI_FLOAT,
                    0,1002,MPI_COMM_WORLD);
            }
        }
    }
//---------------- SLAVE ----------------------------------------
```

Διαφορετικά, αν το κόμβος είναι ένας από τους σκλάβους, δηλαδή αν rank!=0, τότε λαμβάνει από οποιοδήποτε tag τα στοιχεία του πίνακα, υπολογίζει την απαλοιφή Gauss και στέλνει το αποτέλεσμα στον master.

```
MPI_Finalize();
```

Τέλος, με την λειτουργία `MPI_Finalize` ολοκληρώνεται το περιβάλλον εκτέλεσης του MPI.

```
void moirasma(int ntotal,int nmachine,int a[])
{
    int i;
    int nperiseuma,ntask;
    ntask=ntotal/nmachine;
    for(i=0;i<=nmachine-1;i++)
    {
        a[i]=ntask;
    }
    nperiseuma=ntotal-ntask*nmachine;
    for(i=0;i<=nperiseuma-1;i++)
    {
        a[i]++;
    }
}
```

Εκτός κυρίως προγράμματος υλοποιείται η συνάρτηση "void moirasma" που αρχικοποιήθηκε στην αρχή του προγράμματος, σύμφωνα με την οποία μοιράζονται οι γραμμές του πίνακα για να σταλούν στους σκλάβους. Συγκεκριμένα, γίνεται ένα αρχικό μοίρασμα των γραμμών του πίνακα, ntask=ntotal/nmachine, όπου ntask=αριθμός γραμμών που θα αποσταλούν, ntotal=συνολικός αριθμός γραμμών και nmachine=αριθμός παράλληλων μηχανών. Στη συνέχεια, οι περισσευούμενες γραμμές που πρέπει να αποσταλούν, υπολογίζονται με την nperiseuma=ntotal-ntask*nmachine, όπου nperiseuma=αριθμός περισσευούμενων γραμμών από το αρχικό μοίρασμα. Η διαδικασία αυτή εκτελείται για την ορθή εκτέλεση του μοιράσματος. Για παράδειγμα, έστω ότι έχουμε 4 επεξεργαστές και έναν πίνακα 30 γραμμών, στο πρώτο μοίρασμα η κάθε μηχανή θα λάβει 7 γραμμές και οι υπόλοιπες 2 θα σταλούν στο δεύτερο μοίρασμα.

2.2.6 Σύγκριση απόδοσης σειριακού και παράλληλου αλγορίθμου Gauss

Στο σημείο αυτό θα συγκρίνουμε το χρόνο εκτέλεσης του σειριακού και του παράλληλου αλγορίθμου αντίστοιχα. Παρακάτω παραθέτουμε τις τιμές που χρησιμοποιήσαμε για τους υπολογισμούς μας, τις οποίες θα θεωρήσουμε δεδομένες για τις περαιτέρω αναλύσεις μας.

$1Byte = 8bit$

Αν διαθέτουμε *Switch* ταχύτητας $1Gbit/\sec = 1.000.000.000bit/\sec \approx 100MB/\sec$, τότε, δεδομένου ότι ένας αριθμός *float* καταλαμβάνει $4Bytes$, έχουμε

$2Ndata \Rightarrow 8NBytes$ (για τη μεταφορά N αριθμών από και προς τους slaves), δηλαδή

$T_{\mu\varepsilon\tau\alpha\varphi o\rho\acute{\alpha}\varsigma} = \frac{8N}{10^8}\sec$. Αν στο άμεσο μέλλον διαθέτουμε *Switch* ταχύτητας $10Gbit/\sec$, ο χρόνος ελαττώνεται κατά μια τάξη μεγέθους. Στα επόμενα θα θέσουμε ως τιμή αναφοράς για τη μεταφορά δεδομένων το ρυθμό $10^{-7} - 10^{-8}\sec/datum$.

Μετά από χρονομέτρηση στο cluster υπολογίσαμε ότι ένας πολλαπλασιασμός είναι τάξης μεγέθους $10^{-8}\sec$.
Επομένως, ισχύει χονδρικά ότι *χρόνος μεταφοράς/datum* ≈ *χρόνος επεξεργασίας/datum.*

Για την επίλυση του γραμμικού συστήματος Gauss σειριακά, χρειάστηκαν

πράξεις: $\frac{N^3}{3} * 10^{-8}\sec = \left[\frac{N^3}{3} + O\left(N^2\right)\right] * 10^{-8}\sec$

με δεδομένα: $N*(N+1) = N^2 + N \approx N^2$

Για την παράλληλη επίλυση του αλγορίθμου ισχύουν τα παρακάτω:

Η μεταφορά των δεδομένων στο δίκτυο, κατά την οποία διακινούνται η Γραμμή-οδηγός + άλλες γραμμές απαιτεί:

$(send)\ N*(N+1)$ $1^{ος}$ γύρος

$(receive)\ (N-1)(N+1)$

Σύνολο: $N^2+N+N^2-1^2 \approx 2N^2$

$2(N-1)^2$ $2^{ος}$ γύρος

$2(N-2)^2$ $3^{ος}$ γύρος

.

.

.

$2*1^2$ $N^{ος}$ γύρος

διακινήθηκαν συνολικά $2\left(N^2+(N-1)^2+(N-2)^2+...+1^2\right)\approx\frac{2N^3}{3}$ μεταφορές

πράξεις: $\frac{N^3}{3*M}$, όπου M = μηχανήματα

Έστω ότι Χρόνος πράξης: $\Delta t_{πράξης}$

Χρόνος μεταφοράς: $\Delta t_{μεταφοράς}$

Σειριακά: $T_{ολ}=\frac{N^3}{3}\Delta t_{πράξης}$

Παράλληλα: $T_{ολ}'=\frac{2N^3}{3}\Delta t_{μεταφοράς}+\frac{N^3}{3M}\Delta t_{πράξης}$

Παρατηρούμε από τους παραπάνω υπολογισμούς ότι $T_{ολ}\approx T_{ολ}'$, δηλαδή οι χρόνοι είναι συγκρίσιμοι, διότι στη παράλληλη επίλυση επικρατεί πολύ συχνή μεταφορά δεδομένων στο δίκτυο. Για να είναι συμφέρουσα η παράλληλη επίλυση ενός τέτοιου συστήματος θα πρέπει να ισχύει $T_{ολ}' << T_{ολ}$

$$\frac{2N^3}{3}\Delta t_{μεταφοράς}+\frac{N^3}{3M}\Delta t_{πράξης} << \frac{N^3}{3}\Delta t_{πράξης} \Rightarrow 2\Delta t_{μεταφοράς}+\frac{\Delta t_{πράξης}}{M} << \Delta t_{πράξης}$$

$$\Rightarrow \Delta t_{μεταφοράς} << \frac{1}{2}\left(1+\frac{1}{M}\right)\Delta t_{πράξης}$$

π.χ. για $M=4$: $\Delta t_{μεταφοράς} << \frac{5}{8}\Delta t_{πράξης}$

για μεγάλα $M >> 1$, βρίσκουμε ασυμπτωτικά $\Delta t_{μεταφοράς} << \frac{1}{2}\Delta t_{πράξης}$

Διαπιστώνουμε ότι η σειριακή επίλυση του αλγορίθμου είναι προτιμότερη αν $\Delta t_{πράξης} < \Delta t_{μεταφοράς}$, ενώ η παράλληλη επίλυση συμφέρει μόνο αν $\Delta t_{μεταφοράς} << \frac{1}{2}\Delta t_{πραξης}$.

Στην παράλληλη επεξεργασία μεταφέραμε πολλά δεδομένα σε κάθε γύρο, με αποτέλεσμα ο χρόνος μεταφοράς του μεγάλου όγκου των δεδομένων καθώς και ο χρόνος των πράξεων να είναι ασύμφοροι σε σχέση με το χρόνο υπολογισμού του προβλήματος από έναν υπολογιστή. Όπως βλέπουμε στο παραπάνω παράδειγμα μόνο όταν έχουμε πολύ γρήγορο δίκτυο συμφέρει η παράλληλη επεξεργασία.

2.3 Αλγόριθμος Jacobi

2.3.1 Μαθηματική περιγραφή

2.3.1.1 Γενική επαναληπτική μέθοδος

Έστω ένα γραμμικό σύστημα $n * n$

$$Ax = b \tag{1}$$

όπου υποθέτουμε ότι ο πίνακας A είναι ομαλός, δηλαδή το σύστημα έχει μία μοναδική λύση x. Για να ορίσουμε μια επαναληπτική μέθοδο επίλυσης του συστήματος (1), γράφουμε πρώτα τον πίνακα A σαν διαφορά δύο πινάκων

$$A = Q - P \tag{2}$$

όπου ο Q είναι ομαλός, εύκολα αντιστρέψιμος, και θεωρούμε το ισοδύναμο σύστημα

$$(Q - P)x = b$$

ή

$$x = Q^{-1}Px + Q^{-1}b$$

ή

$$x = Cx + d \tag{3}$$

όπου θέτουμε

$$C = Q^{-1}Px \text{ και } d = Q^{-1}b$$

Η επαναληπτική μέθοδος ορίζεται τότε από την επαναληπτική εξίσωση

$$x_k = Cx_{k-1} + d, k = 1,2,... \tag{4}$$

2.3.1.2 Μέθοδος Jacobi

Έστω ένα γραμμικό σύστημα $n * n$

$$Ax = b$$

όπου υποθέτουμε ότι ο πίνακας A είναι ομαλός και ικανοποιεί

$$a_{ii} \neq 0, i = 1,...,n$$

Στην επαναληπτική μέθοδο Jacobi διαλέγουμε τους πίνακες P και Q.

$$Q=\begin{bmatrix} a_{11} & & 0 \\ & a_{22} & \\ & \cdots & \\ 0 & & a_{nn} \end{bmatrix}, Q^{-1}=\begin{bmatrix} \frac{1}{a_{11}} & & 0 \\ & \frac{1}{a_{22}} & \\ & \cdots & \\ 0 & & \frac{1}{a_{nn}} \end{bmatrix}$$

$$P=Q-A=\begin{bmatrix} 0 & -a_{12} & \cdots & -a_{1n} \\ -a_{21} & 0 & & \\ \cdot & & \cdots & \\ \cdot & & \cdots & \\ \cdot & & \cdots & \\ -a_{n1} & & & 0 \end{bmatrix}$$

Η μέθοδος γράφεται τότε

$$x_k = Cx_{k-1} + d, k = 1,2,...$$

x_0 δοθέν, όπου

$$C = Q^{-1}P, d = Q^{-1}b$$

ή

$$x_k=\begin{bmatrix} 0 & -\frac{a_{12}}{a_{11}} & \cdots & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & \cdots & -\frac{a_{2n}}{a_{22}} \\ \cdot & & & \\ \cdot & & & \\ \cdot & & & \\ -\frac{a_{n1}}{a_{nn}} & \cdot & \cdot & 0 \end{bmatrix} x_{k-1} + \begin{bmatrix} \frac{b_1}{a_{11}} \\ \cdot \\ \cdot \\ \cdot \\ \frac{b_n}{a_{nn}} \end{bmatrix}$$

ή ακόμα, θέτοντας $x_k = (x_{1k},...,x_{nk})$

$$x_{ik} = \frac{1}{a_{ii}}(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_{j,k-1}), i = 1,...,n$$

Ας σημειωθεί εδώ ότι κάθε επαναληπτική μέθοδος δίνει αποτελέσματα εφόσον συγκλίνει, δηλαδή η επαναληπτική διαδικασία συγκλίνει προς τη λύση του συστήματος. Η σύγκλιση εξαρτάται από τις ιδιοτιμές του πίνακα C (π.χ. Press et al., 1992). Στα επόμενα θεωρούμε ότι έχουμε εξασφαλίσει τη σύγκλιση του πίνακα.

2.3.2 Σειριακός αλγόριθμος

```
#include <stdio.h>

#define n 1000

int main()
{
    float a[n][n+1],b[n][n],c[n],xnew[n],x[n],sum;
    int i,j,istep;

    for(i=0;i<n;i++)
        for(j=0;j<n+1;j++)
        {
            printf("Dose stoixeio a[%d][%d]",i,j);
            scanf("%f",&a[i][j]);
        }
        for(i=0;i<n;i++)
            for(j=0;j<n+1;j++)
            {
                b[i][j]=-a[i][j]/a[i][i];
                if(i==j) b[i][j]=0;
            }
            for(i=0;i<n;i++)
            {
                c[i]=a[i][n];
            }
            x[0]=0;
            x[1]=0;
            x[2]=0;
            for (istep=1;istep<=100;istep++)
            {
                for(i=0;i<=n-1;i++)
                {
                    sum=0;
                    for(j=0;j<=n-1;j++)
                    {
                        sum+=b[i][j]*x[j];
                    }
                    xnew[i]=sum+c[i];
                }
                for(i=0;i<n;i++)
                {
                    x[i]=xnew[i];
                }
                printf("%d %f %f %f\n", istep,x[0],x[1],x[2]);
            }
}
```

2.3.3. Στρατηγική παραλληλοποίησης του αλγόριθμου Jacobi

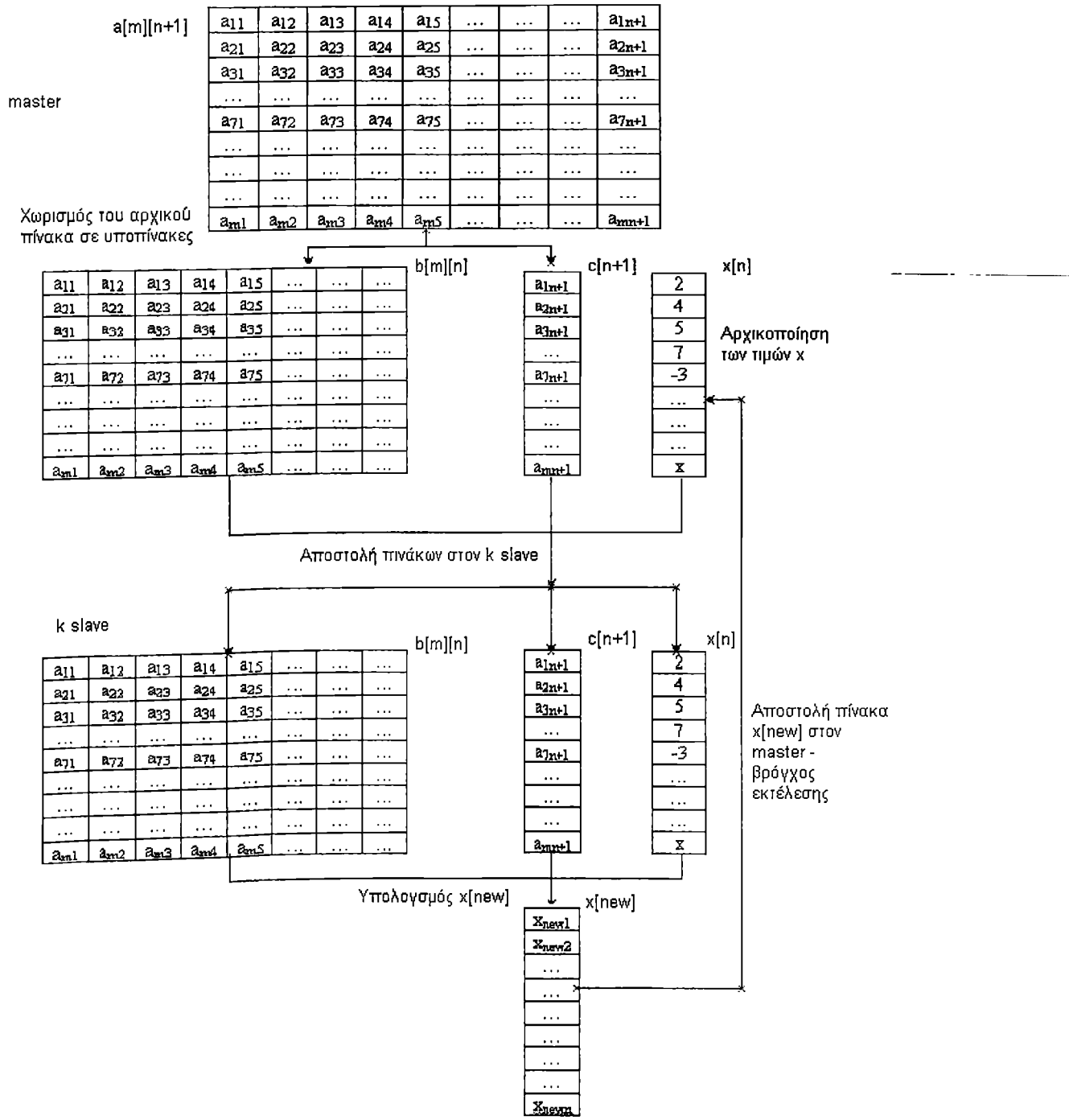
Έστω ότι έχουμε τον πίνακα `a[n][n+1]`.

α) Αν ο αριθμός του μηχανήματος είναι 0 (`rank==0`), δηλαδή είναι ο master, τότε στέλνει σε κάθε κόμβο τον ίδιο αριθμό στοιχείων του πίνακα για τον υπολογισμό του Jacobi για κάθε διαδικασία. Αρχικά ο πίνακας `a[n][n+1]` χωρίζεται σε δύο υποπίνακες `b[n][n]` και `c[n]` και αρχικοποιείται η τιμή των `x[0], x[1], x[2]=0`.

β) Στη συνέχεια ο master στέλνει σε όλους τους άλλους κόμβους του γκρουπ τους πίνακες `b` και `c`, με την εντολή `MPI_Bcast`. Καλείται η συνάρτηση moirasma και υπολογίζει την αντιστοιχία των στοιχείων που θα σταλούν σε κάθε έναν από τους slaves.

γ) Στέλνει στους slaves τον αριθμό των στοιχείων.

δ) Οι slaves υπολογίζουν και στέλνουν την τιμή του `x[n]` στον master.
ε) Αντίστοιχα ο master, αφού λάβει την τιμή του `x[n]` από τους slaves, αντικαθιστά τη νέα τιμή, και επαναλαμβάνει την αποστολή του `x[icount]`, με διαδοχικές επαναλήψεις.
στ) Αντίθετα οι slaves λαμβάνουν την τιμή του `x[icount]` και υπολογίζουν τη νέα τιμή. ζ) Τέλος, τυπώνεται μήνυμα στην οθόνη από τον master για τις τελικές τιμές των `x[0]`, `x[1]`, `x[2]` οι οποίες συγκλίνουν στη λύση του προβλήματος.



Σχήμα 2.2 – Στρατηγική Παραλληλοποίησης αλγόριθμου Jacobi

2.3.4 Παράλληλος αλγόριθμος – Περιγραφή

```
#include "../mpich2-install/include/mpi.h"
#include <stdio.h>
#include <stdlib.h>

void moirasma(int ntotal,int nmachine,int a[]);

#define n 1000

int main(int argc,char *argv[])
{

    float a[n][n+1],b[n][n],c[n],xnew[n],x[n],sum;
    int i,j,istep,numtasks,rc,rank,mpi_any_tag,tag,
        mpi_any_source,itask,pmachine[10],icount,ngrammwn,pmach2;

    MPI_Status Stat;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==0)
    {
        for(i=0;i<n;i++)
            for(j=0;j<n+1;j++)
            {
                printf("Dose stoixeio a[%d][%d]",i,j);
                scanf("%f",&a[i][j]);
            }
            for(i=0;i<n;i++)
                for(j=0;j<n+1;j++)
                {
                    b[i][j]=-a[i][j]/a[i][i];
                    if(i==j)
                        b[i][j]=0;
                }
                for(i=0;i<n;i++)
                {
                    c[i]=a[i][n];
                }
                x[0]=0;
                x[1]=0;
                x[2]=0;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(b,n*n,MPI_FLOAT,0,MPI_COMM_WORLD);
    MPI_Bcast(c,n,MPI_FLOAT,0,MPI_COMM_WORLD);

    if(rank==0)
    {
        printf("prin tin apostoli stoixeiwn stous slaves\n");

        moirasma(n,numtasks,pmachine);
        for (istep=1;istep<=100;istep++)
        {
            icount=pmachine[0];
```

```
            for(itask=1;itask<=numtasks-1;itask++)
            {
                rc=MPI_Send(&icount,1,MPI_INT,itask,
                    1000,MPI_COMM_WORLD);
                rc=MPI_Send(&pmachine[itask],1,MPI_INT,
                    itask,1001,MPI_COMM_WORLD);
                if(pmachine[itask]>0)
                {
                    rc=MPI_Send(&x[0],n,MPI_FLOAT,itask,
                        1002,MPI_COMM_WORLD);
                }
                icount+=pmachine[itask];
            }
            for(i=0;i<=pmachine[0]-1;i++)
            {
                sum=0;
                for(j=0;j<=n-1;j++)
                {
                    sum+=b[i][j]*x[j];
                }
                xnew[i]=sum+c[i];
            }
            for(i=0;i<=pmachine[0]-1;i++)
            {
                x[i]=xnew[i];
            }
            icount=pmachine[0];
            for(itask=1;itask<=numtasks-1;itask++)
            {
                if(pmachine[itask]>0)
                {
                    rc=MPI_Recv(&x[icount],pmachine[itask],
                        MPI_FLOAT,itask,1003,MPI_COMM_WORLD,&Stat);
                }
                icount+=pmachine[itask];
            }
            printf("%d:x[0]=%f,x[1]=%f,x[2]=%f\n",istep,x[0],x[1],x[2]);
        }
    }
    else
    {
        for(istep=1;istep<=100;istep++)
        {
            rc=MPI_Recv(&icount,1,MPI_INT,0,1000,MPI_COMM_WORLD,&Stat);
            rc=MPI_Recv(&pmach2,1,MPI_INT,0,1001,MPI_COMM_WORLD,&Stat);
            if(pmach2>0)
            {
                rc=MPI_Recv(&x[0],n,MPI_FLOAT,0,
                    1002,MPI_COMM_WORLD,&Stat);
                for(i=icount;i<=icount+pmach2-1;i++)
                {
                    sum=0;
                    for(j=0;j<=n-1;j++)
                    {
                        sum+=b[i][j]*x[j];
                    }
                    xnew[i]=sum+c[i];
                }
                for(i=icount;i<=icount+pmach2-1;i++)
                {
                    x[i]=xnew[i];
                }
                rc=MPI_Send(&x[icount],pmach2,MPI_FLOAT,
                    0,1003,MPI_COMM_WORLD);
            }
        }
    }
    MPI_Finalize();
    return 0;
}
```

```
void moirasma(int ntotal,int nmachine,int a[])
{
    int i;
    int nperiseuma,ntask;

    ntask=ntotal/nmachine;

    for(i=0;i<=nmachine-1;i++)
    {
        a[i]=ntask;
    }
    nperiseuma=ntotal-ntask*nmachine;
    for(i=0;i<=nperiseuma-1;i++)
    {
        a[i]++;
    }
}
```

Αρχικά ορίζεται η βιβλιοθήκη που χρησιμοποιείται για τη δημιουργία του προγράμματος. Η οδηγία #include οδηγεί τον μεταγλωττιστή της C να προσθέσει τα συστατικά ενός αρχείου συμπερίληψης στο συγκεκριμένο πρόγραμμα κατά την μεταγλώττιση.

```
#include "../mpich2-install/include/mpi.h"
#include <stdio.h>
#include <stdlib.h>
```

Αρχικοποίηση της συνάρτησης void moirasma. Αυτό επιτυγχάνεται με την δήλωση της στην αρχή του προγράμματος.

```
void moirasma(int ntotal,int nmachine,int a[]);
```

Εδώ χρησιμοποιούμε ακριβώς την ίδια συνάρτηση που αναλύσαμε στην παράγραφο 2.2.4 για τον αλγόριθμο Gauss.

Καθορισμός των στοιχείων των πινάκων, χρησιμοποιώντας την εντολή `#define`.

```
#define n 1000
```

Αρχή κυρίως προγράμματος με την εντολή int main( ).

```
int main(int argc,char *argv[])
```

Δήλωση των μεταβλητών που χρησιμοποιούνται στο πρόγραμμα.

```
float a[n][n+1],b[n][n],c[n],xnew[n],x[n],sum;
int i,j,istep,numtasks,rc,rank,mpi_any_tag,tag,
    mpi_any_source,itask,pmachine[10],icount,ngrammwn,pmach2;
```

Αρχικοποίηση περιβάλλοντος εκτέλεσης του MPI. Καθορίζεται ο αριθμός των διαδικασιών στην ομάδα που συνδέεται με τον πληροφοριοδότη καθώς επίσης και ο βαθμός της καλούμενης διαδικασίας.

```
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Αν ο αριθμός του μηχανήματος είναι 0 (`rank==0`), δηλαδή είναι ο master, τότε στέλνει σε κάθε node τον ίδιο αριθμό στοιχείων του πίνακα για τον υπολογισμό του Jacobi για κάθε διαδικασία.

Επίσης, προκειμένου να ελεγχθούν τα αριθμητικά αποτελέσματα, επιτυγχάνεται γέμισμα του πίνακα από τη σύνταξη του προγράμματος. Αυτό μπορεί να επιτευχθεί και με την πληκτρολόγηση τυχαίων αριθμών από τον χρήστη.

```
if(rank==0)
{
    for(i=0;i<n;i++)
        for(j=0;j<n+1;j++)
        {
            printf("Dose stoixeio a[%d][%d]",i,j);
            scanf("%f",&a[i][j]);
        }
```

Ο πίνακας `a[n][n+1]` χωρίζεται σε δύο υποπίνακες `b[n][n]` και `c[n]` και αρχικοποιείται η τιμή των `x[0], x[1], x[2]=0`.

```
for(i=0;i<n;i++)
    for(j=0;j<n+1;j++)
    {
        b[i][j]=-a[i][j]/a[i][i];
        if(i==j)
            b[i][j]=0;
    }
    for(i=0;i<n;i++)
    {
        c[i]=a[i][n];
    }
    x[0]=0;
    x[1]=0;
    x[2]=0;
```

Δημιουργία φραγμού συγχρονισμού με την χρήση της εντολής `MPI_Barrier`. Κάθε διαδικασία, με την κλήση της `MPI_Barrier`, φράζει μέχρι να φτάσουν όλες οι διαδικασίες του γκρουπ στο ίδιο σημείο κλήσης της `MPI_Barrier`.

```
MPI_Barrier(MPI_COMM_WORLD);
```

Στη συνέχεια ο master στέλνει σε όλα τα άλλα node του γκρουπ τους πίνακες b και c, με την εντολή MPI_Bcast.

```
MPI_Bcast(b,n*n,MPI_FLOAT,0,MPI_COMM_WORLD);
MPI_Bcast(c,n,MPI_FLOAT,0,MPI_COMM_WORLD);
```

Έπειτα καλείται η συνάρτηση `moirasma` και υπολογίζει την αντιστοιχία των στοιχείων που θα σταλούν σε κάθε έναν από τους slaves. Στέλνει στους slaves τον αριθμό των στοιχείων.

```
if(rank==0)
{
    printf("prin tin apostoli stoixeiwn stous slaves\n");

    moirasma(n,numtasks,pmachine);
    for (istep=1;istep<=100;istep++)
    {
        icount=pmachine[0];
        for(itask=1;itask<=numtasks-1;itask++)
        {
            rc=MPI_Send(&icount,1,MPI_INT,itask,
                1000,MPI_COMM_WORLD);
            rc=MPI_Send(&pmachine[itask],1,MPI_INT,
                itask,1001,MPI_COMM_WORLD);
```

Αν το node είναι ένας από τους slaves, με `itask>0`, τότε στέλνει το `x[n]`.

```
if(pmachine[itask]>0)
{
    rc=MPI_Send(&x[0],n,MPI_FLOAT,itask,
        1002,MPI_COMM_WORLD);
}
icount+=pmachine[itask];
```

Αντίστοιχα ο master, αφού λάβει την τιμή του x[n] από τους slaves, αντικαθιστά τη νέα τιμή, η οποία ορίζεται `xnew[n]`, και επαναλαμβάνει την αποστολή του `x[icount]`, με διαδοχικές επαναλήψεις. Αντίθετα οι slaves λαμβάνουν την τιμή του `x[icount]` και υπολογίζουν τη νέα τιμή. Επιπρόσθετα τυπώνεται μήνυμα στην οθόνη από τον master για τις τελικές τιμές των `x[0], x[1], x[2]` οι οποίες συγκλίνουν στη λύση του προβλήματος.

```
      for(i=0;i<=pmachine[0]-1;i++)
      {
          sum=0;
          for(j=0;j<=n-1;j++)
          {
              sum+=b[i][j]*x[j];
          }
          xnew[i]=sum+c[i];
      }
      for(i=0;i<=pmachine[0]-1;i++)
      {
          x[i]=xnew[i];
      }
      icount=pmachine[0];
      for(itask=1;itask<=numtasks-1;itask++)
      {
          if(pmachine[itask]>0)
          {
              rc=MPI_Recv(&x[icount],pmachine[itask],
                  MPI_FLOAT,itask,1003,MPI_COMM_WORLD,&Stat);
          }
          icount+=pmachine[itask];
      }
      printf("%d:x[0]=%f,x[1]=%f,x[2]=%f\n",istep,x[0],x[1],x[2]);

    }
}
  else
  {
      for(istep=1;istep<=100;istep++)
      {
          rc=MPI_Recv(&icount,1,MPI_INT,0,1000,MPI_COMM_WORLD,&Stat);
          rc=MPI_Recv(&pmach2,1,MPI_INT,0,1001,MPI_COMM_WORLD,&Stat);
          if(pmach2>0)
          {
              rc=MPI_Recv(&x[0],n,MPI_FLOAT,0,
                  1002,MPI_COMM_WORLD,&Stat);
              for(i=icount;i<=icount+pmach2-1;i++)
              {
                  sum=0;
                  for(j=0;j<=n-1;j++)
                  {
                      sum+=b[i][j]*x[j];
                  }
                  xnew[i]=sum+c[i];
              }
              for(i=icount;i<=icount+pmach2-1;i++)
              {
                  x[i]=xnew[i];
              }
              rc=MPI_Send(&x[icount],pmach2,MPI_FLOAT,
                  0,1003,MPI_COMM_WORLD);
          }
```

```
        }
    }
```

Με τη λειτουργία `MPI_Finalize` ολοκληρώνεται το περιβάλλον εκτέλεσης του MPI.

```
    MPI_Finalize();
    return 0;
}
```

Τέλος γίνεται το μοίρασμα των στοιχείων με την βοήθεια της συνάρτησης `void moirasma` η οποία αρχικοποιήθηκε στην αρχή του προγράμματος.

```
void moirasma(int ntotal,int nmachine,int a[]);
{
    int i;
    int nperiseuma,ntask;

    ntask=ntotal/nmachine;

    for(i=0;i<=nmachine-1;i++)
    {
        a[i]=ntask;
    }
    nperiseuma=ntotal-ntask*nmachine;
    for(i=0;i<=nperiseuma-1;i++)
    {
        a[i]++;
    }
}
```

2.3.5 Πολυπλοκότητα του αλγορίθμου Jacobi

Στον αλγόριθμο Jacobi οι πολλαπλασιασμοί που γίνονται είναι της τάξης $N*N=N^2$ επί τον αριθμό των επαναλήψεων του βρόγχου που απαιτούνται ώστε να συγκλίνει το τελικό αποτέλεσμα Έτσι η πολυπλοκότητα του αλγόριθμου Jacobi είναι της τάξης $O(N^2)$. Όμως, το πλήθος των απαιτούμενων επαναλήψεων για τη σύγκλιση είναι απροσδιόριστο, και εξαρτάται από τις ιδιοτιμές του πίνακα C και από την απόσταση της αρχικής συνθήκης από τη λύση.

2.3.6 Ανάλυση σχετικής απόδοσης του σειριακού και παράλληλου αλγορίθμου Jacobi

Στο υποκεφάλαιο αυτό θα αναλύσουμε τους χρόνους απόδοσης του σειριακού και παράλληλου αλγόριθμου Jacobi. Τα γραμμικά συστήματα που θα εφαρμοστεί ο Jacobi είναι συστήματα που προκύπτουν από τις διαφορικές εξισώσεις.

Ο αρχικός πίνακας C θεωρείται δεδομένος για όλα τα μηχανήματα, δηλαδή έχουν αντίγραφο οπότε δεν χρειάζεται να γίνει αποστολή από τον master.

Σειριακά: Έστω ότι απαιτούνται Q γύροι για να συγκλίνει

$$x' = -\frac{1}{7}x_1 * x_2 \qquad \text{(2 πολλαπλασιασμοί)} \qquad \left.\begin{matrix} N-1 \\ N-1 \\ \cdot \\ \cdot \\ \cdot \\ N-1 \end{matrix}\right\} N(N-1) = N^2 - N \approx N^2$$

Σύνολο πράξεων: $Q * N^2$ πράξεις

Παράλληλα: $\frac{QN^2}{M}$ πράξεις

μεταφορά: N στοιχεία (τα x') σε κάθε γύρο: $Q * N$

Σειριακά: $\mathrm{T}_{ολ} = QN^2 \Delta t_{πράξης}$

Παράλληλα: $\mathrm{T}_{ολ}' = \frac{QN^2}{M}\Delta t_{πράξης} + QN\Delta t_{μεταφοράς}$

Για μεγάλα N $\frac{\mathrm{T}_{ολ}'}{\mathrm{T}_{ολ}} \approx \frac{1}{M}$

Αν χρειαζόταν να μεταφέρουμε τον πίνακα τότε θα είχαμε:

$$\mathrm{T}_{ολ}' = \frac{QN^2}{M}\Delta t_{πράξης} + QN\Delta t_{μεταφοράς} + N^2\Delta t_{μεταφοράς} = \left(\frac{Q}{M}\Delta t_{πράξης} + \Delta t_{μεταφοράς}\right)N^2$$

$$\frac{\mathrm{T}_{ολ}'}{\mathrm{T}_{ολ}} = \frac{\frac{Q}{M}\Delta t_{πράξης} + \Delta t_{μεταφοράς}}{Q\Delta t_{πράξης}} = \frac{1}{M} + \frac{\Delta t_{μεταφοράς}}{Q\Delta t_{πράξης}}$$

Πρέπει $\frac{\Delta t_{μεταφοράς}}{Q\Delta t_{πράξης}} << Q$ και αν $\Delta t_{μεταφοράς}$ και $Q\Delta t_{πράξης}$ είναι ίσοι τότε συμφέρει.

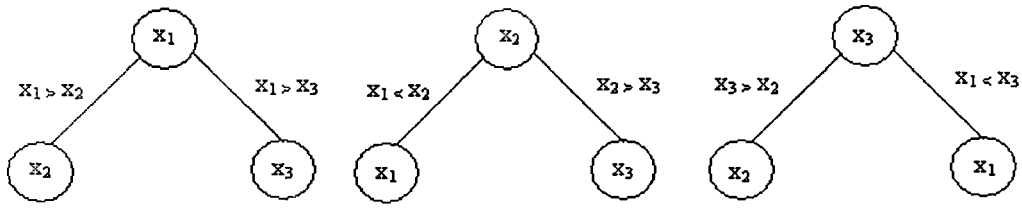
Πρακτικά, με τη μέθοδο Jacobi κερδίζουμε σε χρόνο πάντα με την παραλληλοποίηση, είτε στείλουμε αρχικά τον πίνακα C είτε όχι. Το κέρδος είναι αναλογικά το ίδιο, δηλαδή κατά έναν παράγοντα M, ανεξάρτητα από την ταχύτητα του δικτύου ή των επεξεργαστών.

3. ΑΛΓΟΡΙΘΜΟΣ ΤΑΞΙΝΟΜΗΣΗΣ HEAPSORT

3.1 Περιγραφή αλγορίθμου Heapsort

Στο παρόν κεφάλαιο θα εξετάσουμε τον αλγόριθμο ταξινόμησης Heapsort, ο οποίος βασίζεται στον αλγόριθμο κατασκευής σωρού (π.χ. Παπαθεοδώρου [1998]). Ελέγχει ένα δέντρο από το τελευταίο επίπεδο προς το υψηλότερο (ρίζα του δέντρου) ανεβάζοντας το μεγαλύτερο στοιχείο του στη κορυφή. Η ακολουθία αυτή επαναλαμβάνεται κατά ένα επίπεδο λιγότερο κάθε φορά, έως ότου ολοκληρωθεί η διαδικασία της ταξινόμησης. Τέλος, η διαδικασία ολοκληρώνεται με διάταξη των κόμβων σε λίστα.

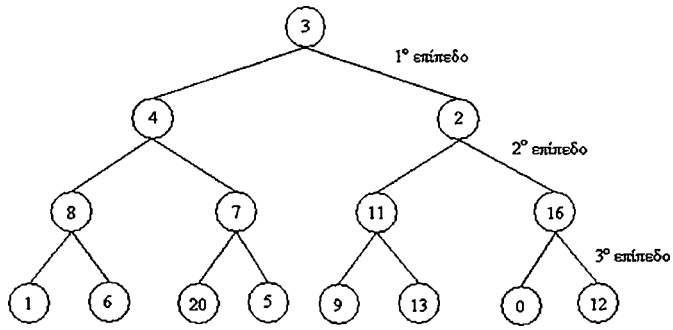
Για παράδειγμα, έστω ότι έχουμε το παρακάτω δέντρο (σχήμα 3.1). Συγκρίνουμε το $x1$ με τα $x2$ και $x3$. Αν $x1 > x2, x3$ τότε δεν θα γίνει καμία εναλλαγή. Αν $x1 < x2$ και $x2 > x3$, τότε θα γίνει εναλλαγή $x1 \leftrightarrow x2$ και το δέντρο διαμορφώνεται όπως στο μεσαίο σχήμα 3.1. Διαφορετικά αν $x1 < x3$ και $x3 > x2$, τότε θα γίνει εναλλαγή $x1 \leftrightarrow x3$ και το δέντρο διαμορφώνεται όπως στο δεξιό σχήμα 3.1.



Σχήμα 3.1

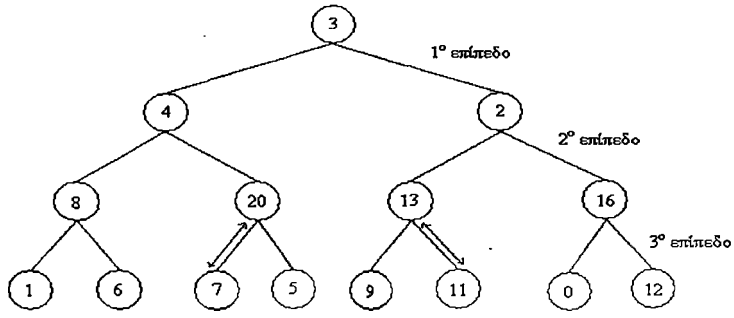
Στη συνέχεια θα περιγράψουμε με διαδοχικά βήματα τον αλγόριθμο Heapsort.

Έστω ότι έχουμε το παρακάτω δέντρο (βλ. σχήμα 3.2



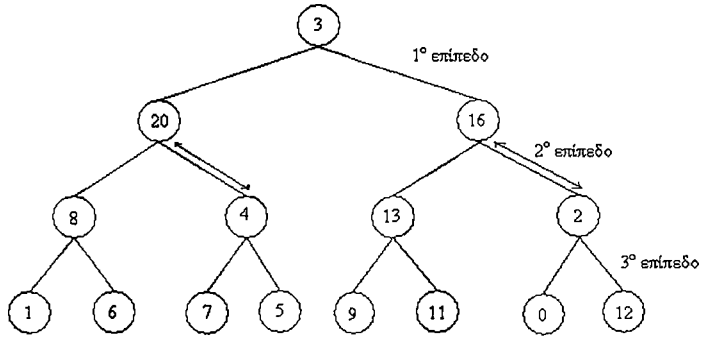
Σχήμα 3.2

Αρχικά θα συγκρίνουμε τη δομή σωρού μέχρι το επίπεδο 3, ξεκινώντας από το τελευταίο επίπεδο, ανεβάζοντας τα μεγαλύτερα στοιχεία των υποδέντρων στις ρίζες τους (σχήμα 3.3).



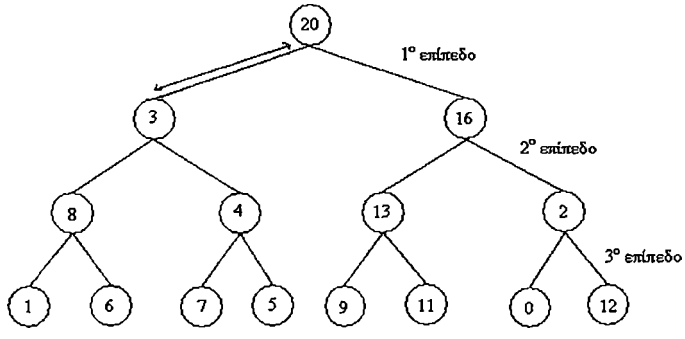
Σχήμα 3.3

Στη συνέχεια συνεχίζουμε τη σύγκριση μέχρι το επίπεδο 2 με την ίδια ακολουθία (σχήμα 3.4).



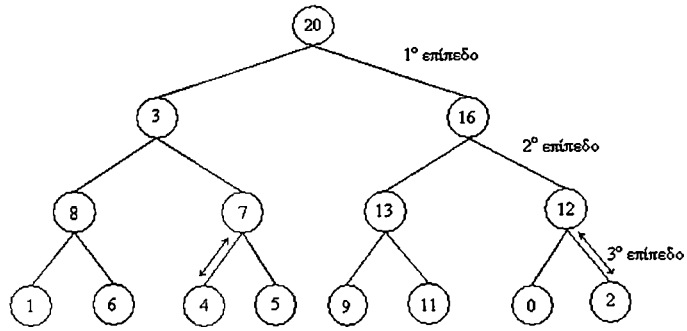
Σχήμα 3.4

Στο τελευταίο στάδιο του πρώτου ελέγχου η σύγκριση ολοκληρώνεται μέχρι το επίπεδο 1, ανεβάζοντας το μεγαλύτερο στοιχείο στην κορυφή (ρίζα) του δέντρου (σχήμα .3.5).

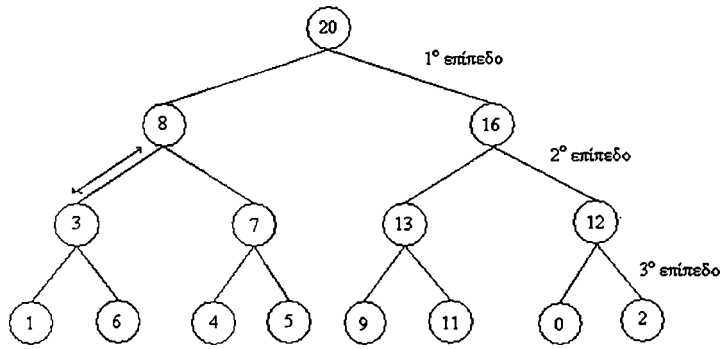


Σχήμα 3.5

Στον δεύτερο έλεγχο των στοιχείων ο αλγόριθμος επαναλαμβάνει τον βρόχο μέχρι το επίπεδο 2 (σχήματα 3.6, 3.7).

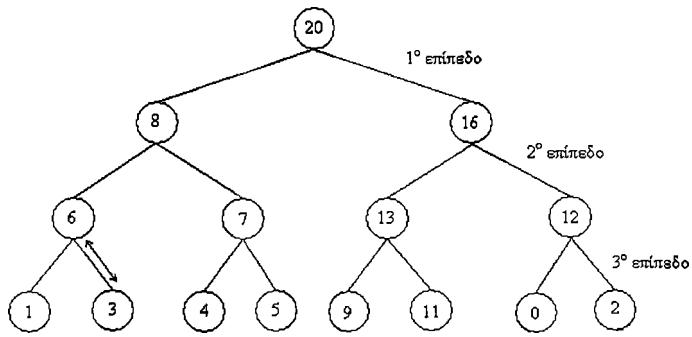


Σχήμα 3.6



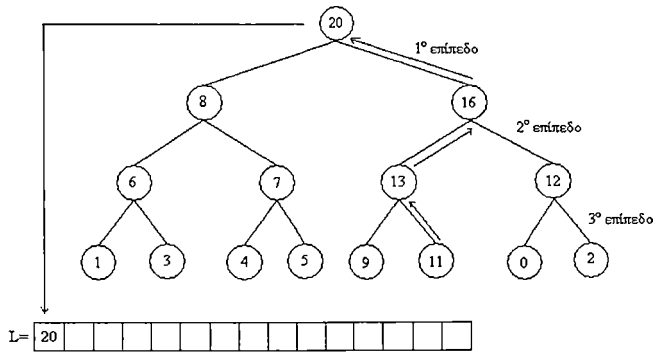
Σχήμα 3.7

Τέλος, εξετάζουμε μόνο το τελευταίο επίπεδο,, και το δέντρο παίρνει την τελική του μορφή (σχήμα 3.8).

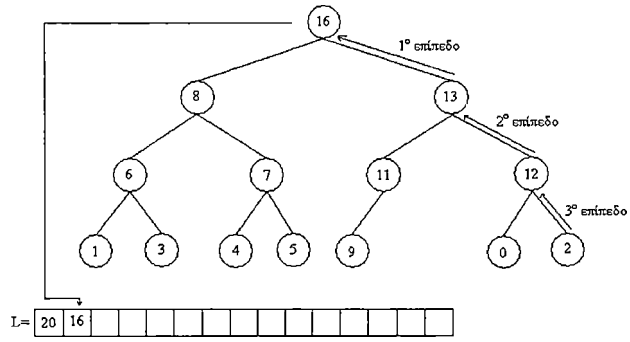


Σχήμα 3.8

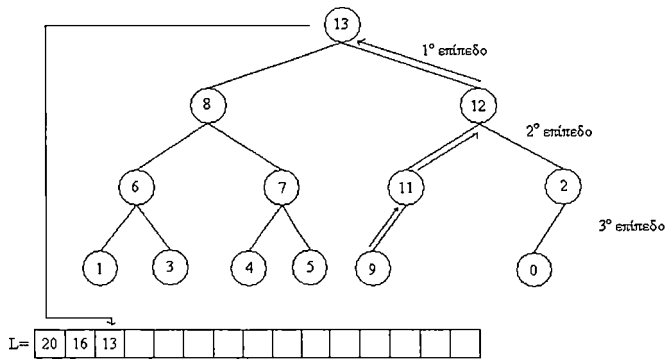
Ο αλγόριθμος στο τελευταίο βήμα του, διατάσσει τα στοιχεία σε λίστα τοποθετώντας το μεγαλύτερο (ρίζα) στην αρχή της λίστας. Καθένα από τα στοιχεία, ανεβαίνοντας επίπεδο, τοποθετούνται στη διάταξη λίστας από το μεγαλύτερο στο μικρότερο. Παρακάτω θα δούμε αναλυτικά τα στάδια της διάταξης (σχήματα 3.9 – 3.23).



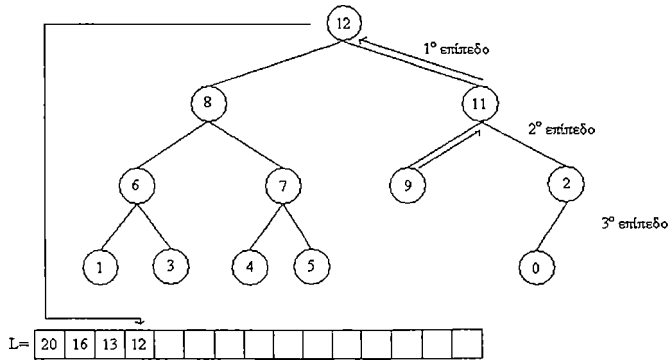
Σχήμα 3.9



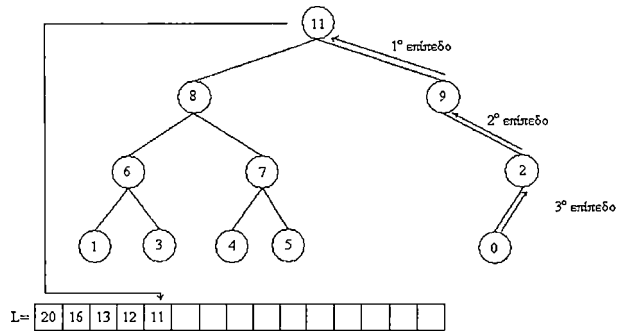
Σχήμα 3.10



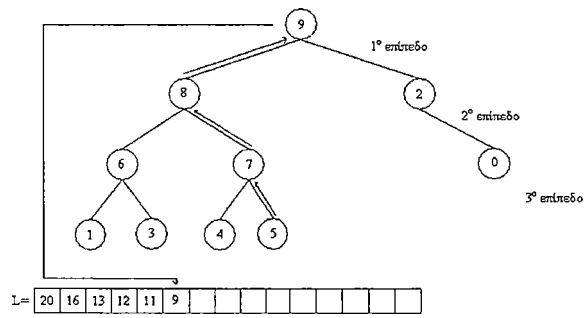
Σχήμα 3.11



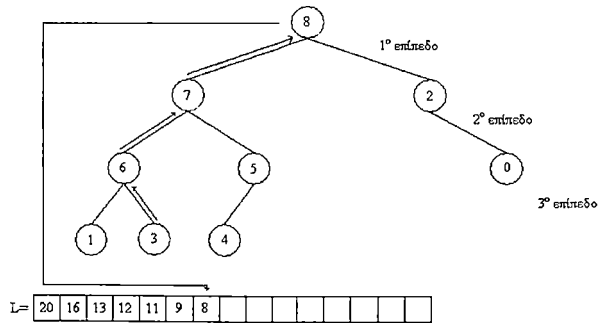
Σχήμα 3.12



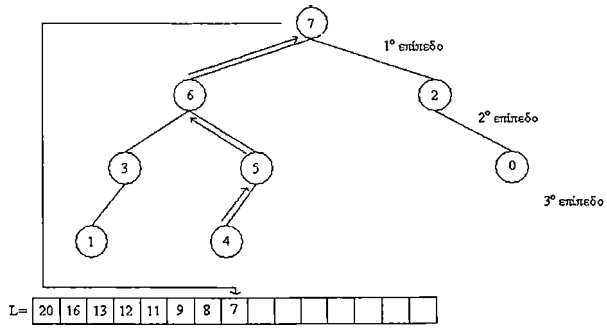
Σχήμα 3.13



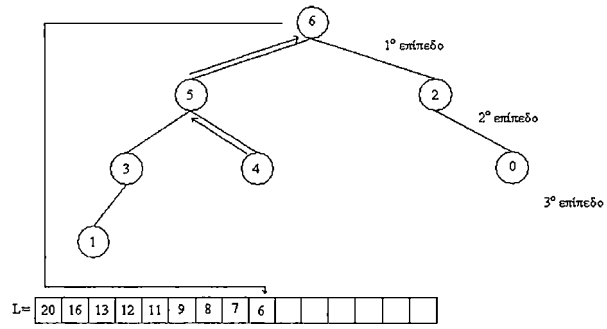
Σχήμα 3.14



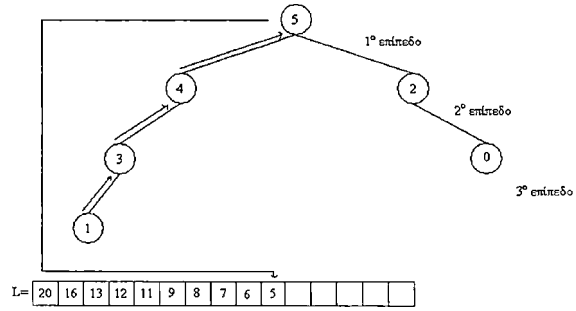
Σχήμα 3.15



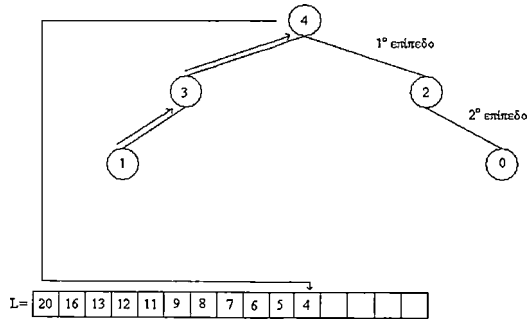
Σχήμα 3.16



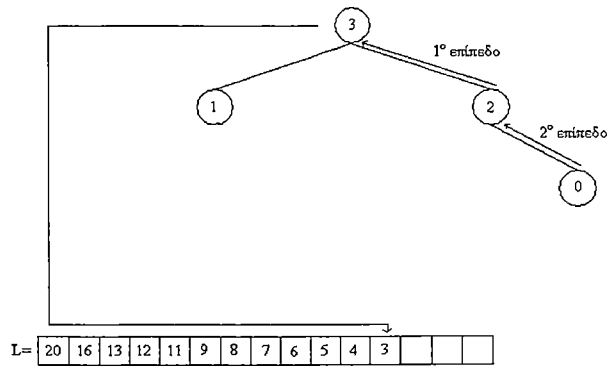
Σχήμα 3.17



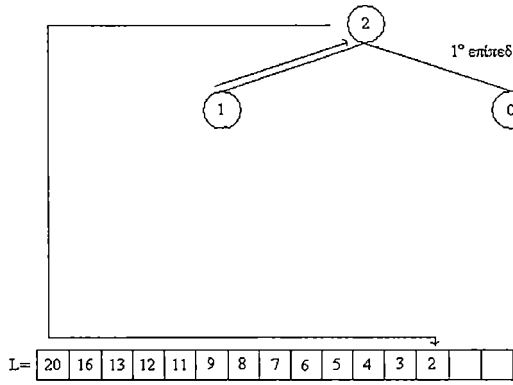
Σχήμα 3.18



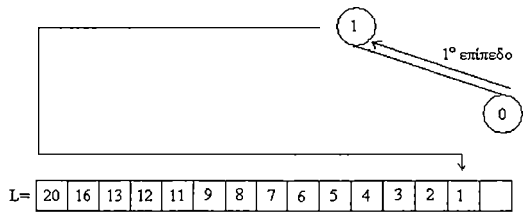
Σχήμα 3.19



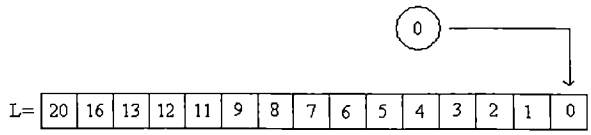
Σχήμα 3.20



Σχήμα 3.21



Σχήμα 3.22



Σχήμα 3.23

3.2 Σειριακός αλγόριθμος

Για την υλοποίηση του αλγορίθμου θεωρούμε παρακάτω ότι ο αριθμός των στοιχείων που πρέπει να ταξινομηθούν είναι δυνάμεις του 2 μείον 1 ($\pi.\chi.\ g = 63 = 2^8 - 1$). Σε αντίθετη περίπτωση, μπορούμε να "γεμίσουμε" τεχνητά τον πίνακα με στοιχεία ίσα, π.χ., με το ελάχιστο στοιχείο του πίνακα, μέχρι το πλήθος των στοιχείων να γίνει ίσο προς την πλησιέστερη δύναμή του δύο μείον ένα, σε σχέση με το αρχικό πλήθος.

Ο σειριακός αλγόριθμος έχει ως εξής:

```
#include <stdio.h>
#include<math.h>
void heapsort(float *a,float *b,float *c);
int mlog(int np);
void heap(float *a, int np);
void taxinomisi(float *a, float *b, int np);

#define n 100

int main()
{
    int i,icount,j,m;
    float a[n],b[n],c[n];

    for(i=0;i<n;i++)
    {
    printf("Dose stoixeia a[%d]", i);
    scanf("%f",&a[i]);
    }


    printf("%d %d\n",mlog(14),mlog(15));
    for(i=0;i<n;i++)
    {
    printf("a[%d]=%f\n", i,a[i]);
    }


    taxinomisi(a,b,n);

    for(i=0;i<n;i++)
    {
        printf("b[%d]=%f\n", i,b[i]);
    }
    printf("icount = %d\n",icount);
}
```

```
//--------------------HEAPSORT--------------------------
void heapsort(float *a,float *b,float *c)
{
    float swap;

    if(*b>*c)
    {
    if(*b>*a)
    {
        swap=*a;
        *a=*b;
        *b=swap;
    }
    }
    else
    {
    if(*c>*a)
    {
        swap=*a;
        *a=*c;
        *c=swap;
    }
    }
}
//---------------------END OF HEAPSORT---------------
//-------------------------Mlog-----------------------
int mlog(int np)
{
    int icount,mod;

    icount=-1;
    mod=1;

    while(mod==1)
    {
    icount++;
    mod=np-(np/2)*2;
    np=np/2;
    }
    return icount;
}
//--------------------END OF Mlog-------------------

//---------------------HEAP-----------------------------
void heap(float *a, int np)
{
    int i,j,k,m,l;
    m=mlog(np);

    for(i=1;i<m;i++)
    {
        printf("i=%d\n", i);
    for(j=m;j>=i;j--)
    {
        for(k=(int)pow(2,(j-1));k<=(int)pow(2,j)-1;k++)
        {
        heapsort(&a[k-1],&a[2*k-1],&a[2*k]);
        }
    }
        for(l=0;l<=np-1;l++)
        {
        printf("a[%d]=%f\n",l,a[l]);
    }
    }
}
//--------------END OF HEAP-----------------------------
```

```
//-----------------TAXINOMISI-------------------------
void taxinomisi(float *a, float *b, int np)
{
    int i,j,k,m,nlim,nmax,icount;
    float min;

    heap(a,np);

    min=a[0];
    for(i=1;i<=np-1;i++)
    {
    if(a[i]<min) min=a[i];
    }

    nlim=(np+1)/2;
    nmax=np-1;
    icount=0;
    for(i=0;i<=np-1;i++)
    {
    if(i==nlim)
    {
        nmax=np-1-nlim;
        nlim=nlim+(np-i+1)/2;
    }
        b[i]=a[0];
    k=0;
    while(2*k+2<=nmax)
        {
        icount++;
        if(a[2*k+1]>a[2*k+2])
        {
        a[k]=a[2*k+1];
        k=2*k+1;
        }
        else
        {
        a[k]=a[2*k+2];
        k=2*k+2;
        }
    }
    a[k]=min;
    }
}

//-----------END OF TAXINOMISI----------------------
```

Στον παραπάνω αλγόριθμο, η ταξινόμηση γίνεται με κλήση της συνάρτησης taxinomisi. Στην παραλληλοποίηση η κλήση αυτή μεταφέρεται σε καθένα από τους επεξεργαστές.

3.3 Στρατηγική παραλληλοποίησης του αλγόριθμου Heapsort

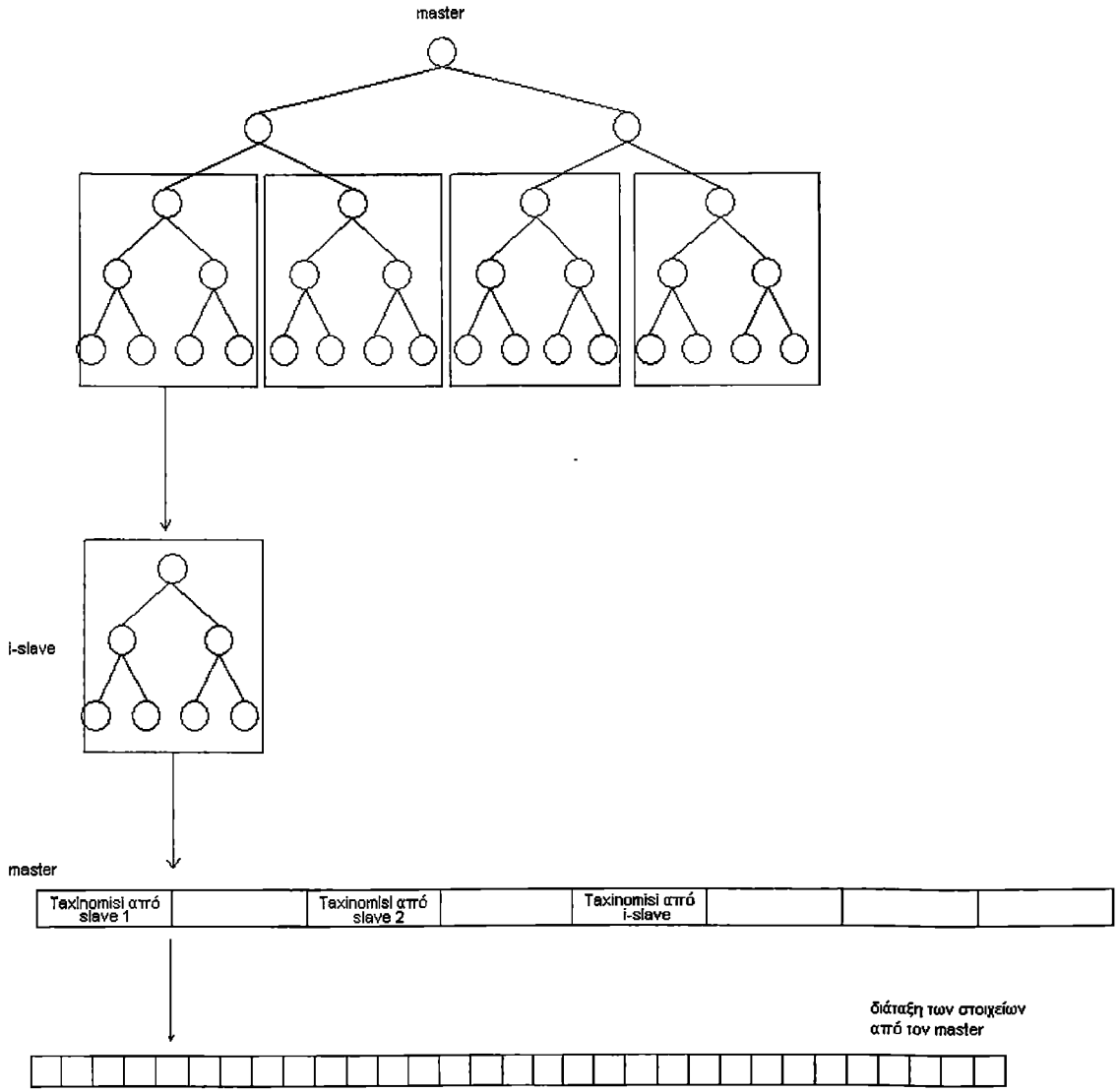
Έστω ότι θέλουμε να παραλληλοποιήσουμε το δέντρο του Σχήματος 6, έχοντας στην διάθεση μας 4 επεξεργαστές (βλ. σχήμα 3.24).

α) Αν ο αριθμός του μηχανήματος είναι $0(rank == 0)$, δηλαδή είναι ο master, και έχοντας στη διάθεση του όλο το δέντρο, θα κρατήσει τα τρία πρώτα στοιχεία και ένα υποδέντρο για τον εαυτό του και στη συνέχεια θα στείλει σε καθέναν από τους slaves τα υποδέντρα που βρίσκονται κάτω από τους κόμβους του επιπέδου 3.

β) Αντίστοιχα οι slaves λαμβάνουν τα υποδέντρα από τον master, ταξινομούν τα στοιχεία εκτελώντας τον αλγόριθμο Heapsort και επιστρέφουν τα ταξινομημένα υποδέντρα.

γ) Στη συνέχεια ο master, αφού λάβει τα ταξινομημένα υποδέντρα, τοποθετεί τα στοιχεία σε πίνακα. Στο σημείο αυτό ο αλγόριθμος εκτελείται σειριακά.

δ) Εφαρμόζει την insertion ώστε να ταξινομηθούν και τα υπόλοιπα στοιχεία που είχε στην κατοχή του.
ε) Τέλος, διατάσσει τα στοιχεία σε λίστα, από το μεγαλύτερο στο μικρότερο.



Σχήμα 3.24 – Στρατηγική παλληλοποίησης αλγορίθμου Heapsort

3.4 Παράλληλος αλγόριθμος – Περιγραφή

```
#include "../mpich2-install/include/mpi.h"
#include <stdio.h>
#include<math.h>
void heapsort(float *a,float *b,float *c);
int mlog(int np);
void heap(float *a, int np);
void taxinomisi(float *a, float *b, int np);

#define n 100

int main(int argc, char *argv[])
{
    int i,j,k,m,nlim,nmax,icount,np,rank,numtasks,rc,itask;
    int mpi_any_tag,tag,mpi_any_source,z,data,nstart,istart,iend,imax;
    float a[n],b[n+1],c[n+1],min,max;
    int ind[100];

    MPI_Status Stat;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    nstart=numtasks-1;
    z=nstart;
    data=(n-(numtasks-1))/numtasks;
    if(rank==0)
    {
        for(i=0;i<n;i++)
        {
            printf("Dose stoixeia a[%d]", i);
            scanf("%f",&a[i]);
        }
        for(itask=1;itask<=numtasks-1;itask++)
        {
            z=z+data;
            rc=MPI_Send(a+z,data,MPI_FLOAT,itask,1000,MPI_COMM_WORLD);
        }

   taxinomisi(a+nstart,b,data);
   z=0;
   for(itask=1;itask<=numtasks-1;itask++)
   {
       z=z+data+1;
       rc=MPI_Recv(b+z,data,MPI_FLOAT,itask,1001,MPI_COMM_WORLD,&Stat);
   }
```

```
 for(i=0;i<n;i++)
 {
     printf("b[%d]=%f\n", i,b[i]);
 }
 for(i=0;i<=data-1;i++)
 {
     iend=(data+1)*i;
     istart=iend+data-1;
     for(j=istart;j>=iend;j--)
     {
         if(a[i]<=b[j])
         {
             b[j+1]=a[i];
             break;
         }
         else
         {
             b[j+1]=b[j];
         }
     }
 }

for(i=0;i<n;i++)
{
    printf("b[%d]=%f\n", i,b[i]);
}
for(i=0;i<=nstart;i++)
{
    ind[i]=(data+1)*i;
}
b[n]=-1.e100;
for(i=0;i<=n;i++)
{
    printf("%d\n",i);
    max=-1.e100;
    imax=0;
    for(j=0;j<=nstart;j++)
    {
        if(ind[j]<=j*(data+1)+data)
        {
            if(b[ind[j]]>max)
            {
                max=b[ind[j]];
                imax=j;
            }
        }
    }
    c[i]=max;
    ind[imax]++;
}
     for(i=0;i<n;i++)
     {
         printf("c[%d]=%f\n", i,c[i]);
     }
 }
 else
 {
     rc=MPI_Recv(a,data,MPI_FLOAT,0,1000,MPI_COMM_WORLD,&Stat);
     taxinomisi(a,b,data);
     rc=MPI_Send(b,data,MPI_FLOAT,0,1001,MPI_COMM_WORLD);
 }


    MPI_Finalize();
    return 0;
}
```

```
//--------------------HEAPSORT-------------------------
void heapsort(float *a,float *b,float *c)
{
    float swap;

    if(*b>*c)
    {
        if(*b>*a)
        {
            swap=*a;
            *a=*b;
            *b=swap;
        }
    }
    else
    {
        if(*c>*a)
        {
            swap=*a;
            *a=*c;
            *c=swap;
        }
    }
}
//----------------------END OF HEAPSORT-------------------

//-------------------------Mlog-----------------------------
int mlog(int np)
{
    int icount,mod;

    icount=-1;
    mod=1;

    while(mod==1)
    {
        icount++;
        mod=np-(np/2)*2;
        np=np/2;
    }
    return icount;
}
//---------------------END OF Mlog-------------------------

//---------------------HEAP-------------------------------
void heap(float *a, int np)
{
    int i,j,k,m;
    m=mlog(np)-1;
//  printf("%d %d\n",np,m);
    for(i=1;i<=m;i++)
    {
        for(j=m;j>=i;j--)
        {
            for(k=(int)pow(2,(j-1));k<=(int)pow(2,j)-1;k++)
            {
//                printf("%d %d %d\n",k-1,2*k-1,2*k);
                heapsort(&a[k-1],&a[2*k-1],&a[2*k]);
            }
        }
    }
}
//--------------END OF HEAP-------------------------------
```

```
//----------------TAXINOMISI--------------------
void taxinomisi(float *a, float *b, int np)
{
    int i,j,k,m,nlim,nmax,icount;
    float min;

    heap(a,np);
    min=a[0];
    for(i=1;i<=np-1;i++)
    {
    if(a[i]<min) min=a[i];
    }

    nlim=(np+1)/2;
    nmax=np-1;
    icount=0;
    for(i=0;i<=np-1;i++)
    {
    if(i==nlim)
    {
        nmax=np-1-nlim;
        nlim=nlim+(np-i+1)/2;
    }
        b[i]=a[0];
    k=0;
    while(2*k+2<=nmax)
        {
        icount++;
        if(a[2*k+1]>a[2*k+2])
        {
        a[k]=a[2*k+1];
        k=2*k+1;
        }
        else
        {
        a[k]=a[2*k+2];
        k=2*k+2;
        }
    }
    a[k]=min;
    }
}

//-----------END OF TAXINOMISI--------------------
```

Αρχικά υπολογίζουμε τα στοιχεία του δέντρου που θα παραμείνουν στον master και αντίστοιχα τα στοιχεία που θα σταλούν στους slaves.

```
nstart=numtasks-1;
z=nstart;
data=(n-(numtasks-1))/numtasks;
```

Αν ο αριθμός του μηχανήματος είναι 0 $(rank == 0)$, δηλαδή είναι ο master, αφού γεμίσει πρώτα ο πίνακας, στέλνει σε κάθε μηχάνημα τα στοιχεία που πρέπει να ταξινομηθούν.

```
if(rank==0)
{

for(i=0;i<n;i++)
{
  printf("Dose stoixeia a[%d]", i);
  scanf("%f",&a[i]);
}

for(itask=1;itask<=numtasks-1;itask++)
{
    z=z+data;
    rc=MPI_Send(a+z,data,MPI_FLOAT,itask,1000,MPI_COMM_WORLD);
}
```

Καλείται η συνάρτηση taxinomisi, λαμβάνει τα στοιχεία από τους slaves και τυπώνει τον πίνακα b[i].

```
taxinomisi(a+nstart,b,data);
z=0;
for(itask=1;itask<=numtasks-1;itask++)
{
    z=z+data+1;
    rc=MPI_Recv(b+z,data,MPI_FLOAT,itask,1001,MPI_COMM_WORLD,&Stat);
}

for(i=0;i<n;i++)
{
    printf("b[%d]=%f\n", i,b[i]);
}
```

Στο σημείο αυτό ο master ενώνει τα στοιχεία που είχε κρατήσει με τα ταξινομημένα data που έλαβε από τους slaves και τυπώνει τον πίνακα.

```
for(i=0;i<=data-1;i++)
{
    iend=(data+1)*i;
    istart=iend+data-1;
    for(j=istart;j>=iend;j--)
    {
    if(a[i]<=b[j])
    {
        b[j+1]=a[i];
        break;
    }
    else
    {
        b[j+1]=b[j];
    }
    }
}

for(i=0;i<n;i++)
{
    printf("b[%d]=%f\n", i,b[i]);
}
```

Τέλος, πραγματοποιεί έλεγχο μεγίστου στοιχείου και τα διατάσσει σε σωρό.

```
for(i=0;i<=nstart;i++)
{
    ind[i]=(data+1)*i;
}

b[n]=-1.e100;
for(i=0;i<=n;i++)
{
    printf("%d\n",i);
    max=-1.e100;
    imax=0;
    for(j=0;j<=nstart;j++)
    {
    if(ind[j]<=j*(data+1)+data)
    {
        if(b[ind[j]]>max)
        {
        max=b[ind[j]];
        imax=j;
        }
    }
    }
    c[i]=max;
    ind[imax]++;

}
for(i=0;i<n;i++)
{
    printf("c[%d]=%f\n", i,c[i]);
}


}
```

Διαφορετικά αν είναι ο slave, με *itask* > 0 λαμβάνει τα data από τον master, ταξινομεί και ξαναστέλνει τα στοιχεία.

```
    else
    {
    rc=MPI_Recv(a,data,MPI_FLOAT,0,1000,MPI_COMM_WORLD,&Stat);
    taxinomisi(a,b,data);
    rc=MPI_Send(b,data,MPI_FLOAT,0,1001,MPI_COMM_WORLD);
    }

   MPI_Finalize();
   return 0;
}
```

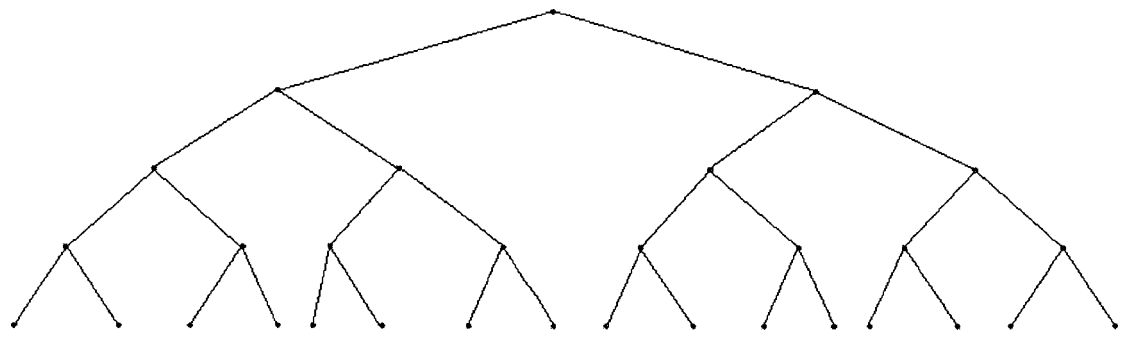
Η συνάρτηση Heapsort ορίζεται για να γίνει η εναλλαγή των στοιχείων του δέντρου και καλείται στην Heap η οποία ελέγχει τα στοιχεία και τα αντιμεταθέτει, εφόσον χρειάζεται. Τέλος, επιτυγχάνεται η ταξινόμηση.

```
//--------------------HEAPSORT---------------------------------
void heapsort(float *a,float *b,float *c)
{
    float swap;

    if(*b>*c)
    {
        if(*b>*a)
   {
            swap=*a;
            *a=*b;
            *b=swap;
        }
    }
    else
    {
        if(*c>*a)
        {
            swap=*a;
            *a=*c;
            *c=swap;
        }
    }
}
//----------------------END OF HEAPSORT----------------------

//-------------------------Mlog-------------------------------
int mlog(int np)
{
    int icount,mod;

    icount=-1;
    mod=1;

    while(mod==1)
    {
        icount++;
        mod=np-(np/2)*2;
        np=np/2;
    }
    return icount;
}
//---------------------END OF Mlog-----------------------------


//---------------------HEAP-------------------------------------
void heap(float *a, int np)
{
    int i,j,k,m;
    m=mlog(np)-1;
//  printf("%d %d\n",np,m);
    for(i=1;i<=m;i++)
    {
        for(j=m;j>=i;j--)
        {
            for(k=(int)pow(2,(j-1));k<=(int)pow(2,j)-1;k++)

            {
//                printf("%d %d %d\n",k-1,2*k-1,2*k);
               heapsort(&a[k-1],&a[2*k-1],&a[2*k]);
        }
        }
    }
}
//--------------END OF HEAP-----------------------------------
```

```
//----------------TAXINOMISI---------------------
void taxinomisi(float *a, float *b, int np)
{
    int i,j,k,m,nlim,nmax,icount;
    float min;

    heap(a,np);
    min=a[0];
    for(i=1;i<=np-1;i++)
    {
    if(a[i]<min) min=a[i];
    }

    nlim=(np+1)/2;
    nmax=np-1;
    icount=0;
    for(i=0;i<=np-1;i++)
    {
    if(i==nlim)
    {
        nmax=np-1-nlim;
        nlim=nlim+(np-i+1)/2;
    }
        b[i]=a[0];
    k=0;
    while(2*k+2<=nmax)
        {
        icount++;
        if(a[2*k+1]>a[2*k+2])
        {
        a[k]=a[2*k+1];
        k=2*k+1;
        }
        else
        {
        a[k]=a[2*k+2];
        k=2*k+2;
        }
    }
    a[k]=min;
    }
}

//-----------END OF TAXINOMISI--------------------
```

3.5 Πολυπλοκότητα αλγόριθμου Heapsort

Στο σημείο αυτό θα αναλύσουμε την πολυπλοκότητα του αλγόριθμου Heapsort.



Έστω ότι έχουμε το παραπάνω δέντρο με 31 στοιχεία. Θεωρώντας ως στοιχειώδη υπολογισμό την τοποθέτηση κάθε τριγώνου του δέντρου σε διάταξη σωρού, στο παραπάνω δέντρο, οι υπολογισμοί που απαιτούνται σε κάθε γύρο είναι:

$1^{ος}$ γύρος → $8+4+2+1$
$2^{ος}$ γύρος → $8+4+2$
$3^{ος}$ γύρος → $8+4$
$4^{ος}$ γύρος → 8

Παρατηρούμε ότι οι τιμές των υπολογισμών είναι δυνάμεις του 2, οπότε για τον πρώτο γύρο εχουμε:

$$8+4+2+1=15 \Leftrightarrow 2^3+2^2+2+1=\frac{2^4-1}{2-1}$$

Για λ υπολογισμούς προκύπτει ότι $\lambda^{\nu}...\lambda^3+\lambda^2+\lambda+1=\frac{\lambda^{\nu+1}-1}{\lambda-1}$

και τα δεδομένα που έχουμε είναι $1+2+2^2+2^3+2^4=\frac{2^5-1}{2-1}=\frac{3^2-1}{1}=31.$

Για N στοιχεία, το πλήθος των τριγώνων $=\frac{N-1}{2}$.

Οπότε οι πράξεις που γίνονται σε κάθε γύρο είναι οι παρακάτω:

$1^{ος}$ γύρος → $\frac{N-1}{2}$

$2^{ος}$ γύρος → $\frac{N-1}{2}-1$

$3^{ος}$ γύρος → $\frac{N-1}{2}-3$

$4^{ος}$ γύρος → $\frac{N-1}{2}-7$

Δηλαδή για $2^{5-1}=31$ νούμερα απαιτούνται 4 γύροι
Ομοίως για $2^{6-1}=63$ νούμερα $\rightarrow 5$ γύροι
$2^{7-1}=127$ νούμερα $\rightarrow 6$ γύροι

Γενικά για N το πλήθος δεδομένα απαιτούνται $g(n)=$γύροι, άρα έχουμε:

$$\log_2(N+1)-1=g(N)$$

Για $g(n)$γύρο έχουμε $\frac{N-1}{2}-2^{g-1}+1$ οπότε οι πράξεις είναι:

$$g\left(\frac{N+1}{2}\right)-2^g+1 \Leftrightarrow \log_2(N+1)*\left(\frac{N+1}{2}\right)-N$$

Επομένως ο αλγόριθμος Heapsort είναι της τάξης $O(N\log N)$, δεδομένου ότι για μεγάλα N ισχύει ότι $\log_2(N+1)\left(\frac{N+1}{2}\right)>>N$.

Σ' ένα παράλληλο σύστημα με m επεξεργαστές όπου $m << N$ το σύνολο των πράξεων που εκτελούνται παράλληλα είναι

$\frac{(N+1)}{2m}\log_2\left(\frac{N+1}{m}\right)-2\frac{N}{m}$, οπότε ο Heapsort έχει πολυπλοκότητα τάξης $O\left(\frac{N\log_2 N}{m}\right)$.

4. ΟΔΗΓΟΣ ΕΓΚΑΤΑΣΤΑΣΗΣ ΕΙΚΟΝΙΚΗΣ ΠΑΡΑΛΛΗΛΗΣ ΜΗΧΑΝΗΣ

Σε αυτό το κεφάλαιο θα παρουσιάσουμε πως γίνεται η εγκατάσταση του MPICH2 σε μια εικονική παράλληλη μηχανή κάτω από το λειτουργικό σύστημα Unix (ή Linux) (Gropp et al, 2004).

Προϋποθέσεις για την εγκατάσταση:

1. Ένα αντίγραφο κατανομής, **mpich2.tar.gz**.
2. Ένα μεταγλωττιστή, Fortran 77, Fortran 90 ή / και μεταγλωττιστής C++ εάν επιθυμούμε να γράψουμε τα προγράμματα MPI σε οποιαδήποτε από αυτές τις γλώσσες.
3. Python 2.2 ή πιο πρόσφατη έκδοση, για την εγκατάσταση του συστήματος διαχείρισης διαδικασίας προεπιλογής MPD. PyXML και XML όπως expat (προκειμένου να χρησιμοποιήσει mpiexec μαζί με MPD). Τα περισσότερα συστήματα έχουν Python, PyXML και expat , εγκατεστημένα από την αρχή.
4. Οποιοδήποτε από τα διάφορα λειτουργικά συστήματα Unix, όπως Linux. Το MPICH2 λειτουργεί στα Linux παρόλα αυτά υπάρχουν μερικά προβλήματα στα συστήματα.

Η παρακάτω διαδικασία εγκατάστασης προεπιλογής εγκαθιστά το MPICH2 έτοιμο για προγράμματα γραμμένα σε C και Fortran –77 χρησιμοποιώντας τη διαδικασία MPD (το οποίο κατασκευάζει και εγκαθιστά το procMpd), χωρίς επιλογές διόρθωσης. Η κατασκευή γίνεται σε ένα τοπικό σύστημα αρχείων, όπου η σύνταξη είναι πιθανό να είναι πολύ γρηγορότερη απ' ότι σε ένα κοινό σύστημα αρχείων. Ο κατάλογος των εγκαταστάσεων που χρησιμοποιείται από τους χρήστες μπορεί να είναι σε ένα κοινό σύστημα αρχείων.

Παρακάτω παρουσιάζονται τα βήματα για τη λήψη του MPICH2 μέσω τρεξίματος του παράλληλου προγράμματος σε παράλληλες μηχανές.

1. Ανοίγουμε το tar αρχείο

tar xfz mpich2.tar.gz

Αν το tar δεν δέχεται την επιλογή z, χρησιμοποιήστε

gunzip –c mpich2.tar.gz | tar xf mpich2.tar

Υποθέτουμε ότι ο κατάλογος όπου κάνουμε αυτό είναι: /home/you/mpilibraries
Περιέχει τώρα ένα subdirectory που ονομάζεται: mpich2-1.0

2. Επιλέγουμε έναν κατάλογο εγκαταστάσεων (ο προεπιλεγμένος κατάλογος είναι /usr/local/bin):

Mkdir/home/you/mpich2 – install

Θα είναι πιο χρήσιμο αν αυτός ο κατάλογος βρίσκεται σε όλες τις μηχανές όπου σκοπεύουμε να τρέξουμε τις διαδικασίες. Αν όχι, θα πρέπει να τον αντιγράψουμε και στις άλλες μηχανές μετά από την εγκατάσταση. Εάν δεν κάνουμε αυτό το βήμα, το επόμενο βήμα θα δημιουργήσει το κατάλογο.

3. Επιλέγουμε ένα κατάλογο κατασκευής. Το κτίσιμο θα ξεκινήσει γρηγορότερα εάν ο κατάλογος κατασκευής είναι σε ένα τοπικό σύστημα αρχείων στη μηχανή όπου γίνονται τα

βήματα διαμόρφωσης και σύνταξης. Αυτός ο κατάλογος είναι απαραίτητο να είναι ξεχωριστός και να παραμείνει αυτούσιος για να μπορεί να χρησιμοποιηθεί και σε άλλες μηχανές.

Mkdir/tmp/you/mpich2 – 1.0

4. Διαμορφώνουμε το MPICH2, διευκρινίζοντας τον κατάλογο εγκαταστάσεων, και το τρέξιμο θα διαμορφώσει τον πρωτότυπο κατάλογο:

cd/tmp/you/mpich2-1.0/home/you/libraries/mpich2-1.0/configure\-
prefix=/home/you/mpich2-install | & tee configure. log

Το παραπάνω εισάγεται σε μία γραμμή.

Ελέγχουμε ότι διαμορφώσαμε το αρχείο ημερολογίου για να σιγουρευτούμε ότι όλα πήγαν καλά.

5. Κατασκευάζουμε το MPICH2:

make | tee make.log

Αυτό το βήμα θα εκτελεστεί σωστά εάν δεν υπήρξε κανένα πρόβλημα με το προηγούμενο. Ελέγχουμε το make.log.

6. Εγκαθιστούμε τις εντολές MPICH2:

make install | & tee install. log

7. Προσθέτουμε το bin subdirectory από την διεύθυνση εγκατάστασης στο δικό μας path:

setenv PATH/home/you/mpich2-install/bin:$PATH

για csh και tcsh
ή
export PATH=/home/you/mpich2-install/bin:$PATH

για bash και sh.

Ελέγχουμε αν είναι στη σωστή θέση οι εντολές:

which mpd
which mpicc
which mpiexec
which mpirun

Όλοι πρέπει να εμπεριέχουν τις εντολές στο bin subdirectory όταν εγκαταστάθηκε ο κατάλογος. Σε αυτό το σημείο είναι απαραίτητο να αντιγραφεί ο κατάλογος και στους υπόλοιπους υπολογιστές εάν δεν βρίσκεται σε ένα κοινό σύστημα αρχείων όπως NFS.

8. Το MPICh2 αντίθετα από το MPICH, χρησιμοποιεί ένα εξωτερικό διευθυντή διαδικασίας για το ξεκίνημα, βήμα προς βήμα, των μεγάλων εργασιών του MPI.
Ο διευθυντής διαδικασίας προεπιλογής καλείται MPD, το οποίο είναι ένα δαχτυλίδι, δαίμονας, στις μηχανές όπου θα τρέξουμε τα προγράμματα MPI. Στα επόμενα βήματα, θα μπορέσουμε να χρησιμοποιήσουμε και να δοκιμάσουμε αυτό το δαχτυλίδι. Ποιο πολλές πληροφορίες για την λειτουργία του MPD θα βρείτε στο αρχείο **README** στο **mpich2/src/pm/mpd**, όπως οι πληροφορίες για το τρέξιμο, η εκτέλεση, πώς να σταματήσει η εκτέλεση και πώς να

χρησιμοποιήσουν το διορθωτή mpigdb. Οι οδηγίες που δίνονται εδώ είναι αρκετές για να αρχίσουμε. Για λόγους ασφαλείας, το mpd ελέγχει το κατάλογο από το αρχείο .mpd.comf με την παρακάτω εντολή

secretword = <secretword>

όπου <secretword> είναι μια λέξη που είναι μόνο σε σας γνωστή. Δεν πρέπει να είναι ο κωδικός πρόσβασης σας στο Unix. Κάνουμε αυτό το αρχείο αναγνώσιμο και επεξεργάσιμο μόνο για μας:

cd $HOME
touch.mpd.conf
cdmod600.mpd.conf
echo "secretword=mr45-j9z">>.mpd.conf

(Βέβαια μπορούμε να χρησιμοποιήσουμε διαφορετική κρυφή λέξη όπως, mr45-j9z)

9. Πρώτος έλεγχος γίνεται για να χρησιμοποιήσει ένα δαχτυλίδι mpd στην τοπική μνήμη, τεστάρει το mpd και το επιστρέφει.

mpd &
mpdtrace
mpdallexit

Η έξοδος του mpdtrace πρέπει να βρίσκετε στο hostname της μηχανής που τρέχετε:
Το mpdallexit αναγκάζει το δαχτυλίδι - δαίμονα mpd να βγει.

10. Τώρα θα χρησιμοποιήσουμε ένα δαχτυλίδι - δαίμονα mpd σε ένα σύνολο επεξεργαστών. Δημιουργούμε ένα αρχείο που αποτελείται από έναν κατάλογο ονομάτων επεξεργαστών, ένα ανά γραμμή. Ονομάζουμε αυτό το αρχείο mpd.hosts.

Τα hostnames θα χρησιμοποιηθούν από το ssh ή rsh, και θα υπάρχουν στα πλήρη ονόματα περιοχών αν κριθεί απαραίτητο. Κάνουμε έλεγχο για να δούμε, αν μπορούμε, τις μηχανές αυτές με το ssh ή rsh χωρίς να δώσουμε προσωπικό κωδικό.
Μπορούμε να δοκιμάσουμε με τα παρακάτω:

ssh othermachine date
ή
rsh othermachine date

Αν δεν μπορέσουμε να το λειτουργήσουμε χωρίς να δώσουμε προσωπικό κωδικό, τότε θα πρέπει να διαμορφώσουμε το ssh ή rsh έτσι ώστε να μπορεί να γίνει, αλλιώς θα κάνουμε τις κατάλληλες εργασίες για το mpdboot, όπως περιγράφονται στο επόμενο βήμα.

11. Ξεκινάμε το δαχτυλίδι - δαίμονα MPD που βρίσκετε στο παρακάτω αρχείο mpd.hosts.

mpdboot –n <number to start> -f mpd.hosts

Ο αριθμός (**number to start**) μπορεί να είναι λιγότερο από 1 + ο αριθμός των hosts στο αρχείο και δεν μπορεί να είναι μεγαλύτερος από 1 + ο αριθμός των hosts στο αρχείο. Ένα mpd αρχίζει να τρέχει στον επεξεργαστή όπου τρέχει το mpdboot. Γίνεται έλεγχος για να δούμε εάν όλοι οι hosts που γράψαμε στο mpd.host είναι σε λειτουργία

mpdtrace

Μετά από αυτό μεταβαίνουμε στο βήμα 12. Εάν δεν μπορέσουμε να εργαστούμε με το mpdboot λόγω δυσκολιών με το ssh ή rsh, τότε μπορούμε να αρχίσουμε το δαχτυλίδι – δαίμονα «με το χέρι» ως εξής:

mpd & # start the local daemon
mpdtrace -1 # makes the local daemon print its host and port in the form
 # <host>_<ports>

Κατόπιν καταγράφουμε σε κάθε μια από τις άλλες μηχανές, το μονοπάτι:

install/bin directory, και mpd –h <hostname> -p <port> &

όπου, hostname και port ανήκουν στο αρχικό mpd, από τότε που ξεκινήσαμε.
Σε κάθε μηχανή, αφού ξεκινήσουμε το mpd, μπορούμε να κάνουμε:

mpdtrace

για να δούμε ποιες μηχανές είναι στο δαχτυλίδι μέχρι τώρα. Περισσότερες πληροφορίες για το mpdboot και άλλες επιλογές για να ξεκινήσουμε το mpd βρίσκονται στο mpich2 – 1.0/src/pm/mpd/README.

12. Εξετάζουμε το δαχτυλίδι - δαίμονα που μόλις δημιουργήσαμε:

mpdtrace

Το αποτέλεσμα θα πρέπει να εμπεριέχει τους hosts που το MPD «τρέχει» αυτή την στιγμή. Μπορούμε να δούμε πόσο γρήγορα διατρέχει το δαχτυλίδι - δαίμονας ένα μήνυμα:

mpdringtest

Αυτό είναι γρήγορο, μπορούμε να δούμε πόσο χρόνο θα πάρει το μήνυμα να διατρέξει το δαχτυλίδι – δαίμονας **(mpdringtest):**

mpdringtest 100
mpdringtest 1000

13. Εξετάζουμε αν το δαχτυλίδι μπορεί να τρέξει μια εργασία multiprocess:

mpdrun –n <number> hostname

Ο αριθμός της διαδικασίας δεν χρειάζεται να ταιριάζει με τον αριθμό των hosts στο δαχτυλίδι, αν υπάρχουν ομοιότητες θα υπάρξει πρόβλημα. Μπορούμε να δούμε τις επιπτώσεις παίρνοντας τα rank labels από το stdout:

mpdrun –1 –n 30 hostname

Πιθανότατα να μην χρειαστεί να δώσουμε ολόκληρο το όνομα του μονοπατιού, αν όχι, θα χρησιμοποιήσουμε το πλήρες όνομα του μονοπατιού:

mpdrun –1 –n 30 /bin/hostname

14. Τώρα μπορούμε να τρέξουμε μια εργασία MPI, χρησιμοποιώντας την εντολή mpiexec, όπως στα πρότυπα MPI-2. Υπάρχουν πολλά παραδείγματα στον κατάλογο εγκατάστασης, τον οποίο έχουμε βάλει ήδη στο μονοπάτι, όπως στο κατάλογο mpich2 – 1.0/examples. Ένα από

αυτά είναι το κλασσικό παράδειγμα cpi, που υπολογίζει το n στην παράλληλη αριθμητική ολοκλήρωση.

mpiexec –n 5 cpi

Όπως με το mpdrun (που χρησιμοποιήσαμε εσωτερικά από το mpiexec), ο αριθμός των διαδικασιών δεν χρειάζεται να αντιστοιχεί με τον αριθμό των hosts. Το παράδειγμα cpi θα μας δείξει ποιοι από τους hosts τρέχουν. Εξ'ορισμού, οι διαδικασίες δουλεύουν η μία μετά την άλλη στους hosts κατά το τρέξιμο μιας εργασίας με το mpiexec. Υπάρχουν πολλές επιλογές για το mpiexec, με τις οποίες μπορούν να τρέξουν πολλαπλές εκτελέσεις, οι hosts μπορούν να διευκρινιστούν (εφ'όσον είναι στο δαχτυλίδι mpd), τα χωριστά επιχειρήματα εντολή – γραμμών και οι μεταβλητές περιβάλλοντος μπορούν να περάσουν σε διαφορετικές διαδικασίες, οι κατάλογοι εργασίας και τα μονοπάτια αναζήτησης των εκτελέσεων μπορούν να διευκρινιστούν.

Εκτελούμε :

mpiexec --help

Ένα χαρακτηριστικό παράδειγμα είναι:

mpiexec –n 1 master: -n 19 slave
ή
mpiexec –n 1 host my machine: -n 19 slave

για να εξασφαλίσουμε ότι η διαδικασία με rank 0 τρέχει στο σταθμό εργασίας μας.

Τα επιχειρήματα μεταξύ ':'s σε αυτή την σύνταξη καλούνται «σύνολα επιχειρήματος», δεδομένου ότι ισχύουν για ένα σύνολο διαδικασιών.

Υπάρχει ένα πρόσθετο επιχείρημα που τίθεται για τα επιχειρήματα που ισχύουν για όλες τις διαδικασίες. Για παράδειγμα, για να πάρουμε τις rank labels στην προκαθορισμένη έξοδο, χρησιμοποιούμε:

mpiexec –default –1 : -n 3 cpi

Η εντολή mpirun από το αυθεντικό MPICH είναι ακόμα διαθέσιμη, αν και δεν υποστηρίζει τόσες επιλογές όσες το mpiexec. Θα το χρησιμοποιήσουμε στην περίπτωση που δεν έχουμε κατατμητή XML για τη χρήση του mpiexec.

Αν έχουν ολοκληρωθεί όλα τα παραπάνω βήματα, τότε έχει εγκατασταθεί επιτυχώς το MPICH2 και μπορούμε να τρέξουμε ένα παράδειγμα MPI.

5. ΒΙΒΛΙΟΓΡΑΦΙΑ

ΒΙΒΛΙΟΓΡΑΦΙΑ

1) Culler, D., 1999: "Parallel Computer Architecture", Morgan Kaufmann, New York.
2) Gropp, W., Lusk, E., Ashton, D., Buntinas, D., Butler, R., Chan, A., Ross, R., Thakur, R., Toonen, B., 2004: "MPICH2 Installer's Guide, Version 0.4, Mathematics and Computer Science Division", Mathematics and Computer Science Division, Argonne National Laboratory, Office of Advanced Scientific Computing Research, SciDAC Program, Office of Science, U.S. Department of Energy.
3) Juhasz, Z., Kacsuk, P., Kranzlmuller, D., 2005: "Distributed and Parallel Systems, cluster and Grid Computing", Springer, New York.
4) Kobler, M., Kim, J., Lilja, D., 1998: "Communication Overhead of MPI, PVM and Sckt Library", University of Minnesota, Minneapolis.
5) Lay, D., 1998: "Linear Algebra and its applications", Addison – Wesley Longman, New York.
6) Pacheco, P., 1997: "PARALLEL PROGRAMMING with MPI", Morgan Kaufmann, New York.
7) Press, W., Flannery, B. Teukolsky, S., Vetterling, W., 1992: "Numerical Recipes in C: The Art of Scientific Computing", Press Syndicate of the University of Cambridge, Cambridge, Massachusetts.
8) Wilkinson, B., Allen, M.,1999: "Parallel Programming", Prentice Hall, Upper Saddle River, New York.

ΕΛΛΗΝΙΚΗ ΒΙΒΛΙΟΓΡΑΦΙΑ

1) Ευθυμιόπουλος, Χ.,2002: "Σημειώσεις Αριθμητικής Ανάλυσης-Προγραμματισμού", Τμήμα Στατιστικής και Αναλογιστικής Επιστήμης, Πανεπιστήμιο Αιγαίου.
2) Μαμάτας Ελευθέριος,: "Μελέτη και υλοποίηση κατανεμημένης (Peer to Peer) εφαρμογής", Δημοκρίτειο Πανεπιστήμιο Θράκης, Ξάνθη.
3) Μπακόπουλος, Α., Χρυσοβέργης, Ι., 1992: "Εισαγωγή στην Αριθμητική Ανάλυση", Εκδόσεις Συμεών, Αθήνα.
4) Παπαθεοδώρου, Θ., 1998: "Αλγόριθμοι, εισαγωγικά θέματα και παραδείγματα", Εκδόσεις Πανεπιστημίου Πατρών, Πάτρα.

ΠΗΓΕΣ INTERNET

1) Lancaster Univesity: http://www.lancs.ac.uk/
2) The Netlib: www.netlib.org
3) CSM: Computer Science and Mathematics Division: http://www.csm.ornl.gov/
4) Information & Communication Technologies: http://www.it.uom.gr/
5) Wikipedia: http://www.wikipedia.org/