ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΤΟ ΜΟΝΤΕΛΟ ΜΕΤΑΒΙΒΑΣΗΣ ΜΗΝΥΜΑΤΩΝ
MESSAGE PASSING INTERFACE (MPI)

Της φοιτήτριας
Αριστέας Χρονοπούλου
Αρ. Μητρώου: 01 / 1736

Επιβλέπων Καθηγητής
Αθανάσιος Μάργαρης

Θεσσαλονίκη 2009

ΠΡΟΛΟΓΟΣ

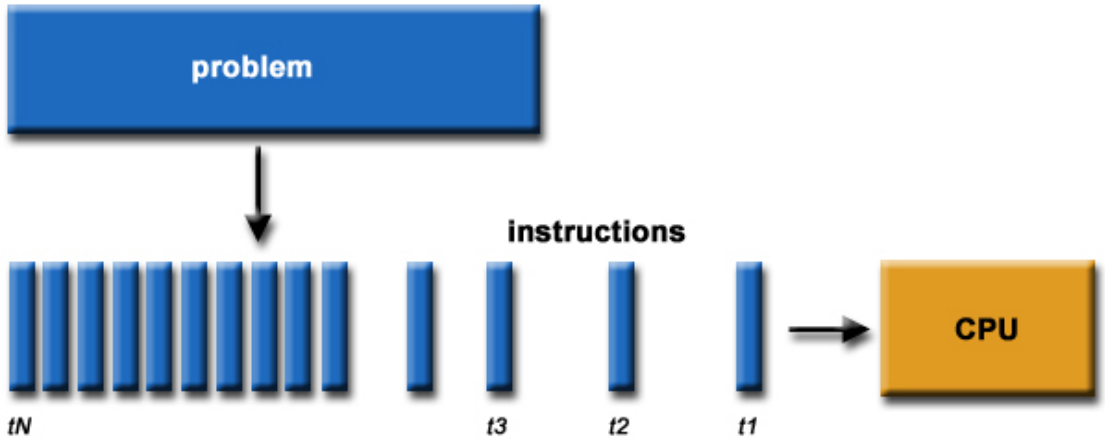
Η παρούσα μελέτη έγινε στα πλαίσια της πτυχιακής εργασίας για το τμήμα Πληροφορικής του Αλεξάνδρειου Τ.Ε.Ι. Θεσσαλονίκης, με επιβλέποντα καθηγητή τον κύριο Αθανάσιο Μάργαρη.
Το θέμα της εργασίας είναι η ανάπτυξη παράλληλων εφαρμογών χρησιμοποιώντας τη βιβλιοθήκη του MPI.
Για την εκπόνησή της απαιτήθηκε τόσο η εκτέλεση των εφαρμογών που παρουσιάζονται σε προγραμματιστικό περιβάλλον MPI, όσο και η συλλογή πληροφοριών για παράλληλα συστήματα και αλγορίθμους, από γενική και ειδική βιβλιογραφία καθώς και από διάφορες διευθύνσεις του διαδικτύου.
Σκοπός της εργασίας είναι η παρουσίαση και η ανάλυση των παράλληλων αλγορίθμων που χρησιμοποιούνται στην εφαρμογή των παράλληλων συστημάτων, καθώς και των τρόπων με τους οποίους μπορούν να αντιμετωπιστούν τα διάφορα προβλήματα που εμφανίζονται.
Η ερευνητική προσέγγιση γίνεται με την εξής σειρά:
Στην αρχή παρουσιάζονται τα παράλληλα υπολογιστικά συστήματα και οι κατηγορίες τους (κεφάλαια 1,2), ενώ αμέσως μετά ακολουθεί η παρουσίαση του πρωτοκόλλου επικοινωνίας MPI (κεφάλαιο 3). Στη συνέχεια αναλύονται οι παράλληλοι αλγόριθμοι με τη χρήση της βιβλιοθήκης του MPI (κεφάλαια 4 – 7) και στο τέλος παρατίθεται η διαδικασία εγκατάστασης του προτύπου MPI (κεφάλαιο 8).

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΕΦΑΛΑΙΟ 1

1. Εισαγωγή στην παράλληλη επεξεργασία

Αρχικά, τα διάφορα πακέτα λογισμικού σχεδιάζονταν με τέτοιο τρόπο ώστε να μπορούν να εκτελούνται σε έναν υπολογιστή που περιλαμβάνει μια επεξεργαστική μονάδα (CPU). Το κάθε πρόβλημα έπρεπε να διασπαστεί σε μια ακολουθία εντολών η οποίες εκτελούνταν σειριακά. Μόνο μια εντολή μπορούσε να εκτελεστεί κάθε στιγμή (Σχήμα 1).

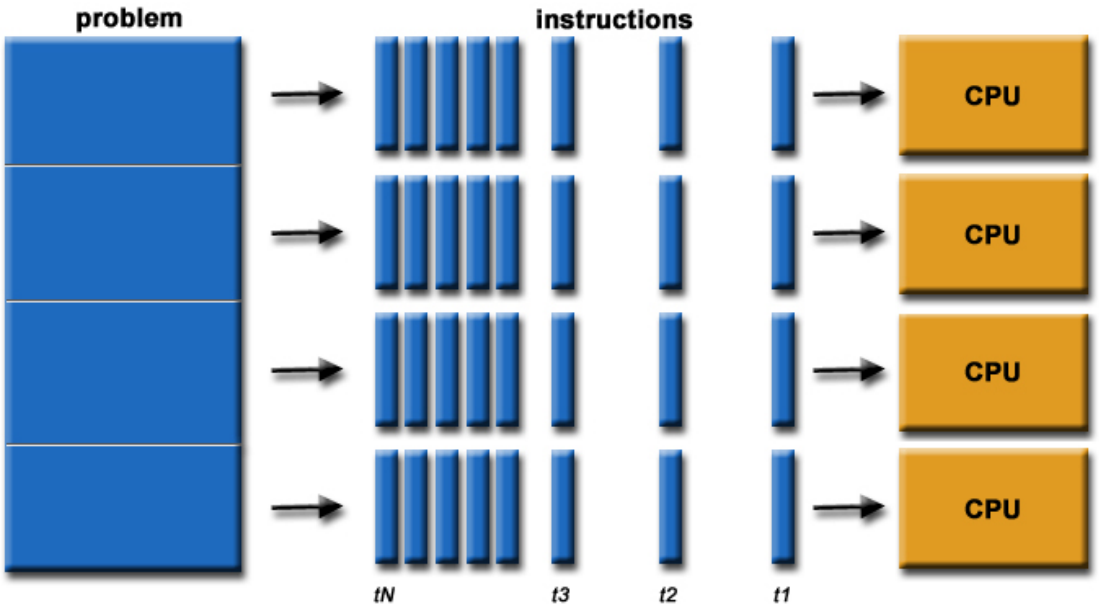


Σχήμα 1. *Σειριακή επεξεργασία*

Καθώς όμως οι ανάγκες για υπολογιστική ισχύ μεγάλωναν, άρχισε σιγά-σιγά να αναπτύσσεται η παράλληλη επεξεργασία. Ως παράλληλη επεξεργασία ορίζουμε την υπολογιστική διαδικασία η οποία εκτελείται σε υπολογιστικά περιβάλλοντα με περισσότερες από μία επεξεργαστικές μονάδες, οι οποίες λειτουργούν και επεξεργάζονται δεδομένα ταυτόχρονα. Η παράλληλη επεξεργασία χρησιμοποιεί πολλούς επεξεργαστές όπου όλοι εργάζονται προς λύση του ιδίου προβλήματος με στόχο την ισομερή κατανομή του υπολογιστικού φόρτου. Το προς επίλυση πρόβλημα διασπάται σε υπο-προβλήματα που μπορούν να επιλυθούν ταυτόχρονα. Κάθε υπο-πρόβλημα διασπάται σε ένα σύνολο εντολών και οι εντολές αυτές εκτελούνται ταυτόχρονα από διαφορετικές υπολογιστικές μονάδες (Σχήμα 2).

Στην ιδανική περίπτωση, αν t_n είναι ο υπολογιστικός χρόνος που απαιτεί μια επεξεργαστική μονάδα για να λύσει ένα πρόβλημα και n ο αριθμός των επεξεργαστικών μονάδων, ο χρόνος που απαιτείται από ένα σύστημα παράλληλης επεξεργασίας για τη λύση του ίδιου προβλήματος είναι ίσος με:

$$t = \frac{t_n}{n}$$



Σχήμα 2. *Παράλληλη επεξεργασία*

Οι υπολογιστικοί πόροι που συμμετέχουν στην παράλληλη επεξεργασία αποτελούνται από ηλεκτρονικούς υπολογιστές με πολλούς επεξεργαστές, από πλήθος ηλεκτρονικών υπολογιστών συνδεδεμένους σε ένα δίκτυο ή από έναν συνδυασμό των παραπάνω.

Η παράλληλη επεξεργασία καθιστά δυνατή την επίλυση πολύπλοκων προβλημάτων που σχετίζονται με διάφορες επιστήμες, όπως:

- Φυσική
- Βιολογία, Γενετική
- Χημεία, Μοριακές επιστήμες
- Γεωλογία, Σεισμολογία
- Εφαρμοσμένη μηχανική
- Μικροηλεκτρονική, Σχεδιασμός κυκλωμάτων
- Μετεωρολογία
- Επιστήμη υπολογιστών, Μαθηματικά
- ...

ΚΕΦΑΛΑΙΟ 2

2. Παράλληλα υπολογιστικά συστήματα

Υπάρχουν πολλοί τύποι υπολογιστικών συστημάτων με δυνατότητες πολυεπεξεργασίας τα οποία μπορούν να χωρισθούν σε κατηγορίες ανάλογα με τον αριθμό των επεξεργαστών, το είδος της μνήμης και τον τρόπο ανταλλαγής μηνυμάτων μεταξύ των επεξεργαστών. Οι κύριες κατηγορίες παράλληλων υπολογιστών, ανάλογα με τον τύπο των βασικών χαρακτηριστικών τους, παρουσιάζονται παρακάτω.

2.1. Αριθμός επεξεργαστικών μονάδων

Ανάλογα με τον αριθμό των επεξεργαστικών μονάδων, τα παράλληλα υπολογιστικά συστήματα διακρίνονται σε:

- *Πυκνής δομής* (fine-grained) ή *μαζικής παραλληλίας* (massively parallel) όταν αποτελούνται από μερικές εκατοντάδες επεξεργαστικές μονάδες
- *Μεσαίας δομής* (medium-grained) όταν αποτελούνται από μερικές δεκάδες επεξεργαστικές μονάδες και
- *Αραιής δομής* (coarse-grained) όταν αποτελούνται από μερικούς επεξεργαστές.

Επειδή ο αριθμός των επεξεργαστικών μονάδων ενός παράλληλου υπολογιστικού περιβάλλοντος υψηλών επιδόσεων μπορεί να αυξομειώνεται, ένα πολύ σημαντικό χαρακτηριστικό των παράλληλων υπολογιστικών συστημάτων είναι η επεκτασιμότητα τους (scalability), δηλαδή η ιδιότητά τους να επιτυγχάνουν γραμμική αύξηση (στην ιδανική περίπτωση) της απόδοσής τους με την αύξηση του αριθμού των επεξεργαστών.

Τα πιο οικονομικά συστήματα παράλληλης επεξεργασίας είναι τα δίκτυα σταθμών εργασίας (workstation networks). Εντούτοις, στην περίπτωση αυτή περιορίζεται η αύξηση της αποτελεσματικότητας του δικτύου με την προσθήκη νέων επεξεργαστικών μονάδων (σταθμοί εργασίας – workstations) εξαιτίας της σχετικά αργής επικοινωνίας μεταξύ των επεξεργαστικών μονάδων. Για να επιτευχθεί ικανοποιητική βελτίωση της απόδοσης του συστήματος θα πρέπει η ανταλλαγή δεδομένων μεταξύ των επεξεργαστικών μονάδων να πραγματοποιείται σε μικρό χρονικό διάστημα σε σχέση με αυτό που απαιτείται για τους τοπικούς υπολογισμούς οι οποίοι πραγματοποιούνται σε κάθε επεξεργαστική μονάδα.

2.2. Αρχιτεκτονικές μνήμης σε παράλληλη επεξεργασία

Κατά παράλληλη επεξεργασία υπάρχουν τρεις διαθέσιμες διαφορετικές αρχιτεκτονικές μνήμης:

- *διαμοιραζόμενη μνήμη (shared memory)*,
- *κατανεμημένη μνήμη (distributed memory) και*
- *υβριδική κατανεμημένη-διαμοιραζόμενη μνήμη (hybrid distributed-shared memory).*

2.2.1. Διαμοιραζόμενη μνήμη (shared memory)

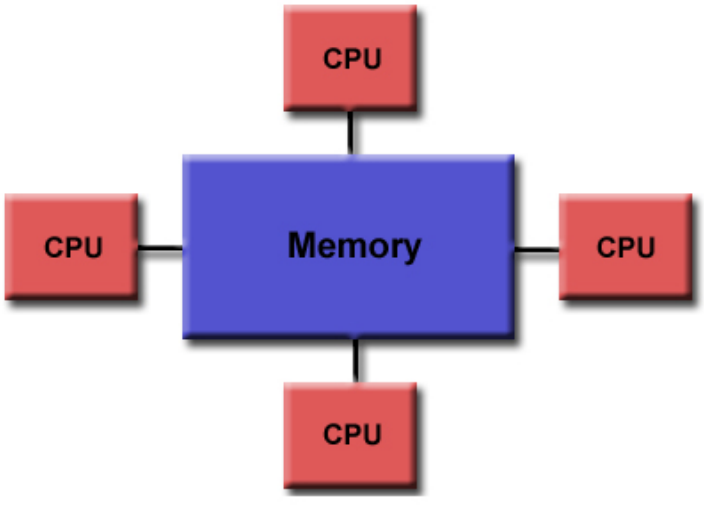
Τα βασικά χαρακτηριστικά της συγκεκριμένης αρχιτεκτονικής είναι τα παρακάτω:

- Όλες οι επεξεργαστικές μονάδες έχουν πρόσβαση σε όλες τις θέσεις μνήμης.
- Οι επεξεργαστικές μονάδες λειτουργούν ανεξάρτητα αλλά μοιράζονται τους ίδιους πόρους σε μνήμη.
- Οι αλλαγές που προκύπτουν στη μνήμη από μια επεξεργαστική μονάδα, είναι ορατές και σε όλες τις υπόλοιπες μονάδες.

Τα υπολογιστικά συστήματα που χρησιμοποιούν διαμοιραζόμενη μνήμη χωρίζονται σε δύο κατηγορίες ανάλογα με τους χρόνους πρόσβασης στη μνήμη: *Uniform Memory Access* (UMA) και *Non-Uniform Memory Access* (NUMA).

Uniform Memory Access (UMA)

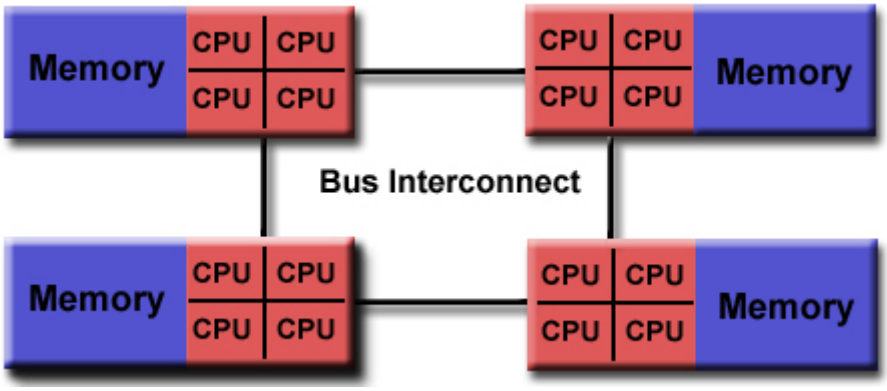
Σύμφωνα με την αρχιτεκτονική UMA, σε κάθε επεξεργαστική μονάδα διατίθεται ίσος χρόνος πρόσβασης στη μνήμη και κάθε φορά που ανανεώνεται η τιμή κάποιας θέσης μνήμης, όλες οι επεξεργαστικές μονάδες ενημερώνονται για την αλλαγή αυτή (Σχήμα 3).



Σχήμα 3. *Shared Memory (UMA)*

Non-Uniform Memory Access (NUMA)

Σύμφωνα με την αρχιτεκτονική NUMA, πολλά SMPs (Symmetric Multiprocessor machines) συνδέονται μεταξύ τους σε φυσικό επίπεδο. Κάθε SMP αποτελείται από ένα πλήθος επεξεργαστικών μονάδων που χρησιμοποιούν μια διαμοιραζόμενη μνήμη. Η μνήμη ενός SMP είναι προσπελάσιμη από οποιοδήποτε άλλο SMP που ανήκει στο σύστημα. Οι χρόνοι πρόσβασης που διατίθενται σε κάθε SMP για κάθε μνήμη δεν είναι ίσοι μεταξύ τους. Η αρχιτεκτονική NUMA παρουσιάζεται στο Σχήμα 4.



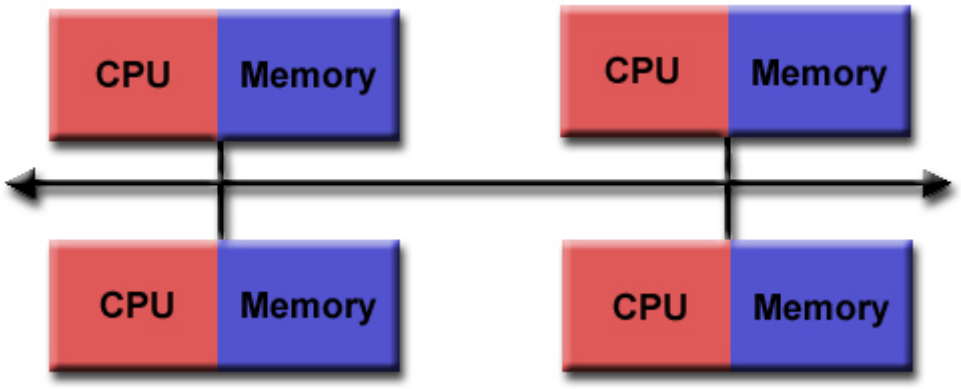
Σχήμα 4. *Shared Memory (NUMA)*

Τα βασικά πλεονεκτήματα της αρχιτεκτονική διαμοιραζόμενης μνήμης είναι αφενός η αποτελεσματικότερη επικοινωνία μεταξύ των επεξεργαστών (αφού δεν έχουμε μετακίνηση δεδομένων από και προς τις τοπικές μνήμες των επεξεργαστών), γεγονός το οποίο συνεπάγεται μεγαλύτερη απόδοση σε σχέση με τα κατανεμημένης μνήμης, και αφετέρου η ευκολότερη συγγραφή και παραγωγή κατάλληλου λογισμικού σε σχέση με αυτή των υπολογιστικών συστημάτων κατανεμημένης μνήμης.

Η αρχιτεκτονική διαμοιραζόμενης μνήμης παρουσιάζει, όμως, και βασικά μειονεκτήματα όπως το μεγάλο κόστος αγοράς καθώς επίσης και η μέτρια προς κακή επεκτασιμότητά τους αφού η προσπέλαση της διαμοιραζόμενης μνήμης γίνεται μέσω διαύλων επικοινωνίας πεπερασμένου εύρους ζώνης (bandwidth). Έτσι, η αύξηση του αριθμού των επεξεργαστικών μονάδων αυξάνει τη συχνότητα προσπέλασης στην διαμοιραζόμενη μνήμη και έχει ως αποτέλεσμα τη συμφόρηση των διαύλων επικοινωνίας και κατά συνέπεια πτώση της απόδοσής τους. Επίσης, ο προγραμματιστής έχει την ευθύνη του συγχρονισμού που θα εξασφαλίσει την ομαλή χρήση της καθολικής μνήμης.

2.2.2. Κατανεμημένη μνήμη (distributed memory)

Το δεύτερο είδος υπολογιστικών συστημάτων είναι ένα είδος στο οποίο κάθε επεξεργαστική μονάδα έχει τη δική της μνήμη την οποία δε μοιράζεται με τις υπόλοιπες (Σχήμα 5). Απλώς ανταλλάσσει δεδομένα με μερικές ή και όλες τις υπόλοιπες επεξεργαστικές μονάδες με τη χρήση κατάλληλων διαύλων.



Σχήμα 5. *Κατανεμημένη μνήμη (distributed memory)*

Υπάρχουν διάφορες τοπολογίες υπολογιστών με κατανεμημένη μνήμη, όπως π.χ. σειράς, δακτυλίου, καννάβου κλπ. Οι παράμετροι οι οποίες χαρακτηρίζουν αυτή την τοπολογία του δικτύου είναι οι εξής:

- Η διάμετρος η οποία ορίζεται ως η μέγιστη απόσταση την οποία πρέπει να διανύσει ένα μήνυμα μεταξύ δύο κόμβων
- Ο βαθμός κάθε κόμβου ο οποίος ορίζεται ως ο αριθμός των συνδέσεων του με άλλους κόμβους

Οι ως άνω παράμετροι είναι καθοριστικής σημασίας για μια τοπολογία δικτύου διότι επηρεάζουν σημαντικά το κόστος και την απόδοσή τους.

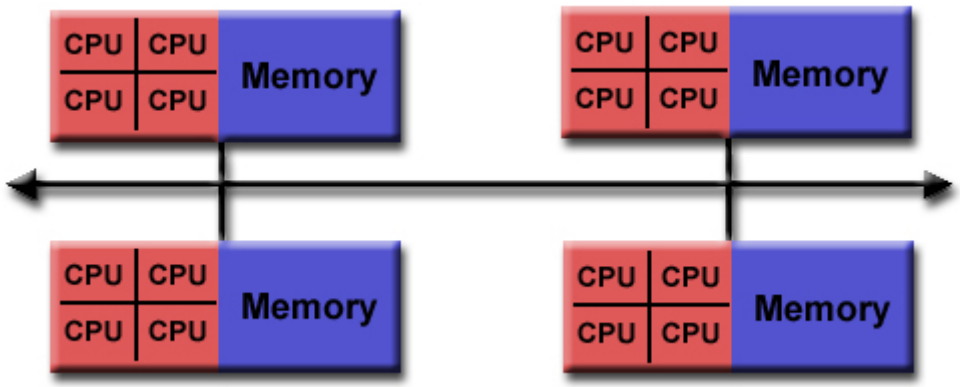
Κάθε επεξεργαστική μονάδα ενεργεί ανεξάρτητα από τις υπόλοιπες, καθώς διαθέτει την δική της μνήμη. Οι αλλαγές που προκαλεί στην δική της μνήμη δεν έχουν καμιά επίδραση στη μνήμη των υπόλοιπων μονάδων. Όταν μια επεξεργαστική μονάδα χρειάζεται πρόσβαση στα δεδομένα μιας άλλης μονάδας, είναι ευθύνη του προγραμματιστή να καθορίσει τον τρόπο της επικοινωνίας αυτής. Η δομή του δικτύου που απαιτείται για την μετάδοση των δεδομένων συχνά ποικίλει, αν και μπορεί να είναι τόσο απλή όσο το Ethernet.

Το μεγάλο πλεονέκτημα της συγκεκριμένης αρχιτεκτονικής είναι ότι μπορεί να εκμεταλλευτεί πολλές επεξεργαστικές μονάδες μικρού κόστους που συνεργάζονται μεταξύ τους. Έτσι, είναι εφικτή η κατασκευή ενός ισχυρού υπολογιστικού συστήματος με σχετικά χαμηλό κόστος με τη συνεχή προσθήκη επεξεργαστικών μονάδων. Σε αυτή την κατηγορία ανήκουν και τα δίκτυα σταθμών εργασίας στα οποία οι υπολογιστές συνδέονται με καλώδια συρμάτινα ή οπτικών ινών και η παράλληλη εκτέλεση των προγραμμάτων γίνεται με τη χρήση κατάλληλου προγραμματιστικού μέσου διασύνδεσης για εφαρμογές (application programming interface – API).

Τα μειονεκτήματα ενός τέτοιου είδους υπολογιστικών συστημάτων αφενός η δυσκολότερη συγγραφή κατάλληλου λογισμικού το οποίο να εκμεταλλεύεται στο έπακρο την επεξεργαστική ισχύ ενός τέτοιου συστήματος και αφετέρου η επικοινωνία μεταξύ των επεξεργαστικών μονάδων είναι πιο αργή σε σχέση με αυτή των συστημάτων διαμοιραζόμενης μνήμης. Αυτό οφείλεται στη χρήση διαύλων μικρότερου εύρους ζώνης και στο γεγονός ότι τα μηνύματα μεταξύ των επεξεργαστικών μονάδων συνήθως περιέχουν πολύ μεγαλύτερη ποσότητα δεδομένων, συγκρινόμενα με αυτά των συστημάτων διαμοιραζόμενης μνήμης, διότι σε αυτά πολλές φορές ενυπάρχουν δεδομένα τα οποία δεν είναι διαθέσιμα σε όλες τις επεξεργαστικές μονάδες.

2.2.3. Υβριδική κατανεμημένη-διαμοιραζόμενη μνήμη (Hybrid distributed-shared memory)

Τα μεγαλύτερα και ταχύτερα υπολογιστικά συστήματα σήμερα χρησιμοποιούν έναν συνδυασμό των αρχιτεκτονικών διαμοιραζόμενης και κατανεμημένης μνήμης (Σχήμα 6).



Σχήμα 6. *Υβριδική κατανεμημένη-διαμοιραζόμενη μνήμη (Hybrid distributed-shared memory)*

Το συστατικό της συγκεκριμένης αρχιτεκτονικής που σχετίζεται με την αρχιτεκτονική διαμοιραζόμενης μνήμης, είναι συνήθως ένα SMP. Οι επεξεργαστές ενός SMP χειρίζονται τη μνήμη του SMP σαν καθολική.

Από την άλλη μεριά, το συστατικό που σχετίζεται με την αρχιτεκτονική κατανεμημένης μνήμης, είναι η δικτύωση πολλών SMPs. Κάθε SMP έχει πρόσβαση μόνο στη δική του μνήμη, επομένως απαιτείται δικτυακή επικοινωνία για να επιτευχθεί μεταφορά δεδομένων από ένα SMP σε κάποιο άλλο.

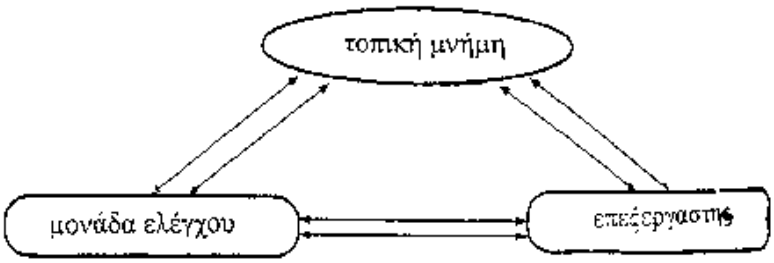
Η υβριδική αρχιτεκτονική κατανεμημένης-διαμοιραζόμενης μνήμης χαρακτηρίζεται από τον συνδυασμό των πλεονεκτημάτων και μειονεκτημάτων των δύο αρχιτεκτονικών από τις οποίες απαρτίζεται.

2.3. Ροή διεργασιών και δεδομένων

Κάθε ηλεκτρονικός υπολογιστής, όπως είναι γνωστό αποτελείται από τρία βασικά συστατικά (σχήμα 7):

- τη μονάδα ελέγχου,
- τη μονάδα υπολογισμών (ή επεξεργαστή) και
- τη μνήμη

Οι παράλληλοι υπολογιστές δεν έχουν απαραίτητα πολλαπλά αντίγραφα των ανωτέρω τμημάτων, εκτός από τη μονάδα υπολογισμών. Όπως ήδη έχουμε αναφέρει παραπάνω, η μνήμη μπορεί να είναι διαμοιραζόμενη ή κατανεμημένη και κατά ανάλογο τρόπο μπορεί να υπάρχουν μια ή πολλαπλές μονάδες ελέγχου οι οποίες να «στέλνουν» οδηγίες (instructions) και δεδομένα (data) στις μονάδες υπολογισμών.



Σχήμα 7. *Τα βασικά συστατικά ενός ηλεκτρονικού υπολογιστή*

Έτσι, σύμφωνα με την ταξινόμηση κατά Flynn, τα είδη παράλληλων αρχιτεκτονικών είναι τα εξής:

- *SISD (Single Instruction-Single Data)* το οποίο είναι το απλό ακολουθιακό μοντέλο υπολογισμών όπου διεξάγεται μία σειρά υπολογισμών ανά ομάδα δεδομένων.

- *SIMD (Single Instruction-Multiple Data)* στο οποίο οι επεξεργαστικές μονάδες εκτελούν τις ίδιες πράξεις με διαφορετικά δεδομένα. Αυτή η αρχιτεκτονική είναι κατάλληλη για παραλληλία μικρού όγκου υπολογισμών και είναι ανεφάρμοστη σε συστήματα με πολλές επεξεργαστικές μονάδες.

- *MISD (Multiple Instruction-Single Data)* το οποίο είναι το αντίστροφο του μοντέλου SIMD το οποίο και αποτελεί απλώς μία θεωρητική δυνατότητα με αδυναμία εφαρμογής σε πρακτικά προβλήματα.

- *MIMD (Multiple Instruction-Multiple Data)* στο οποίο κάθε επεξεργαστική μονάδα έχει την αυτονομία ενός SISD μοντέλου για αυτό και αποτελεί τη μόνη αρχιτεκτονική γενικής εφαρμογής αφού δεν αντιμετωπίζει τα προβλήματα κανονικότητας των δεδομένων και συγχρονισμού της αρχιτεκτονικής SIMD.

Οι συνηθέστερες αρχιτεκτονικές είναι τύπου SIMD ή τύπου MIMD με την επικρατέστερη να είναι τύπου MIMD. Σε αυτό το σημείο αξίζει να σημειωθεί ότι εκτός από τα κανονικά συστήματα πολυεπεξεργασίας, μπορούμε να χρησιμοποιήσουμε ένα δίκτυο υπολογιστών, είτε σειριακών είτε παράλληλων, ως ένα ενιαίο υπολογιστικό περιβάλλον πολυεπεξεργασίας με τη χρήση κατάλληλου API. Τα πιο διαδεδομένα λογισμικά αυτού του τύπου είναι τα PVM (Parallel Virtual Machine) και MPI (Message Passing Interface) με το οποίο και θα ασχοληθούμε εκτενέστερα στο επόμενο κεφάλαιο.

2.4. Λογισμικά ανταλλαγής μηνυμάτων

Διάφορα συστήματα μπορούν να χρησιμοποιηθούν για τη διαχείριση των μηνυμάτων που ανταλλάσσονται κατά την εκτέλεση παράλληλων προγραμμάτων σε ένα διαδικτυωμένο υπολογιστικό σύστημα. Τα συστήματα αυτά παρέχουν ρουτίνες-υποπρογράμματα για την ενεργοποίηση διαδικασιών (tasks or processes) ούτως ώστε οι διαδικασίες αυτές να επικοινωνούν και να συγχρονίζονται μεταξύ τους κατά τη διάρκεια της παράλληλης εκτέλεσης των υπολογισμών. Η επικοινωνία μεταξύ τους γίνεται μέσω διαύλων επικοινωνίας ενώ αν μιλάμε για ένα διαδικτυωμένο υπολογιστικό σύστημα τότε γίνεται μέσω καλωδίων ή οπτικών ινών.
Τα πιο διαδεδομένα συστήματα ανταλλαγής μηνυμάτων είναι τα εξής:

- *Socket interface* ο οποίος παρέχει τη δυνατότητα για κλήσεις συστήματος (system calls) για άμεση πρόσβαση σε υπηρεσίες επικοινωνίας μέσω των πρωτοκόλλων TCP/IP ή UDP.

- *Remote procedure call (RPC)* ο οποίος επενεργεί επί των πρωτοκόλλων TCP/IP ή UDP σε υψηλότερο επίπεδο από το προηγούμενο σύστημα αφού κατά τις αιτήσεις για επικοινωνία μέσω του δικτύου πολλές λεπτομέρειες παραμένουν αδιαφανείς για τον προγραμματιστή. Το σύστημα αυτό εφαρμόζεται κατά τη χρήση του μοντέλου task farming ή master-slave αφού η επικοινωνία RPC λαμβάνει τη μορφή μηνυμάτων αίτησης από τον master κόμβο, ο οποίος ζητά να γίνει κάποια εργασία προς κάποιον ή κάποιους slave κόμβους οι οποίοι διεκπεραιώνουν την εργασία και επιστρέφουν δεδομένα στον master κόμβο.

- *API ανταλλαγής μηνυμάτων (message passing APIs)* τα οποία παρέχουν ένα σύνολο ρουτινών για τις πιο δημοφιλείς και ευρέως χρησιμοποιούμενες γλώσσες προγραμματισμού ούτως ώστε να είναι δυνατή η διαχείριση των απαιτήσεων επικοινωνίας που παρουσιάζουν οι παράλληλες εφαρμογές κατανεμημένης μνήμης. Τα λογισμικά αυτά, όπως π.χ. το PVM ή το MPI, παρουσιάζονται ως οι πιο διαδεδομένες λύσεις για τη διαχείριση των απαιτήσεων επικοινωνίας σε κατανεμημένα λογισμικά ανάλυσης με τη μέθοδο των πεπερασμένων στοιχείων. Η προτίμηση αυτή βασίζεται στο γεγονός ότι αυτά τα λογισμικά προσφέρουν υψηλού βαθμού μεταφερσιμότητα (portability) και στον σχετικά απλό τρόπο

διαχείρισης των απαιτήσεων επικοινωνίας που παρέχονται από τέτοια APIs.

ΚΕΦΑΛΑΙΟ 3

3. Το περιβάλλον προγραμματισμού MPI

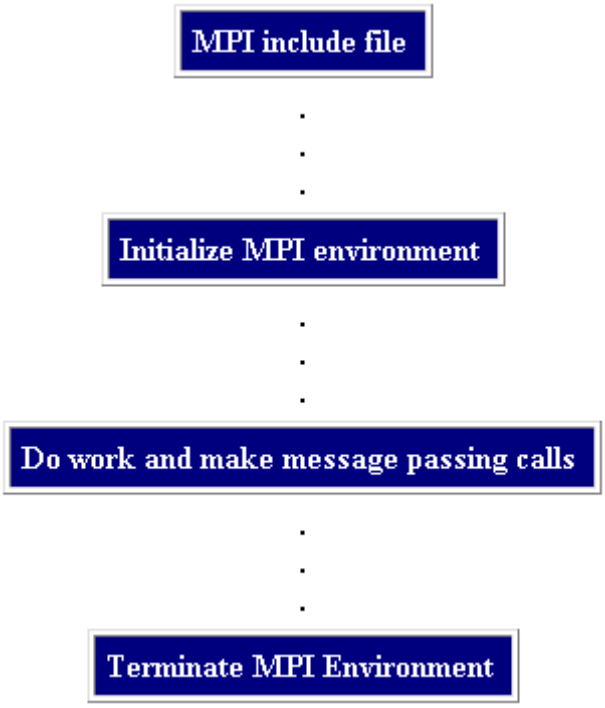
Το περιβάλλον διεπαφής ανταλλαγής μηνυμάτων (message passing interface) MPI είναι ένα πρωτόκολλο επικοινωνίας το οποίο χρησιμοποιείται για τη δημιουργία εφαρμογών που στηρίζονται στις αρχές του παράλληλου προγραμματισμού. Το πρότυπο MPI δεν είναι τίποτα άλλο από μια συλλογή συναρτήσεων που μπορεί να χρησιμοποιηθεί μέσα από προγράμματα που είναι γραμμένα σε γλώσσα C ή σε γλώσσα Fortran και τα οποία επιτρέπουν τη δημιουργία και χρήση ενός πλήθους διεργασιών που εκτελούνται παράλληλα στον ίδιο ή σε διαφορετικούς υπολογιστές και επικοινωνούν μεταξύ τους χρησιμοποιώντας το μοντέλο ανταλλαγής μηνυμάτων.

Το MPI υποστηρίζει επικοινωνίες από σημείο σε σημείο (point to point communication), καθώς και συλλογικές επικοινωνίες (collective communications). Οι στόχοι του είναι η υψηλή απόδοση, η εξελιξιμότητα , και η φορητότητα.

Μέχρι και σήμερα, το MPI παραμένει το κυρίαρχο μοντέλο που χρησιμοποιείται στον υπολογισμό υψηλών επιδόσεων. Χρησιμοποιείται κυρίως σε μοντέλα κατανεμημένης μνήμης (distributed memory), καθώς επίσης και σε μοντέλα διαμοιραζόμενης μνήμης (shared memory).

3.1. Η δομή προγράμματος MPI

Η δομή ενός προγράμματος που χρησιμοποιεί το MPI, παρουσιάζεται στο επόμενο σχήμα (Σχήμα 8). Πρώτα θα πρέπει να περιλάβουμε το header file που μας προσφέρει το MPI. Έπειτα γίνεται η αρχικοποίηση του περιβάλλοντος, στη συνέχεια ακολουθούν οι λειτουργίες που πρόκειται να εκτελεστούν και τέλος γίνεται ο τερματισμός του περιβάλλοντος MPI.



Σχήμα 8. *Δομή προγράμματος MPI*

3.2. Οι δομικές μονάδες του MPI

Διεργασία (process): Η στοιχειώδης μονάδα μιας εφαρμογής MPI είναι η διεργασία (process) η οποία δημιουργείται και εκτελείται ανεξάρτητα από τις υπόλοιπες διεργασίες του συστήματος χρησιμοποιώντας τους δικούς της πόρους (resources).

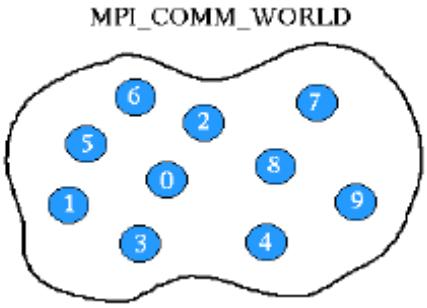
Τάξη (rank): Η κάθε διεργασία ταυτοποιείται με μοναδικό τρόπο από ένα κωδικό διεργασίας (Process Id) που στην ορολογία του προτύπου ονομάζεται τάξη (rank) και λαμβάνει ακέραιες τιμές μεγαλύτερες ή ίσες με το μηδέν. Εάν το πλήθος των διεργασιών που εκτελούνται στο σύστημα είναι γνωστό και ίσο με N, τότε οι κωδικοί αυτών των διεργασιών θα έχουν τις N συνεχόμενες τιμές 0,1,2,3,….,N-1.
Επειδή οι τάξεις των διεργασιών προσδιορίζουν τις διεργασίες του συστήματος με μοναδικό τρόπο, χρησιμοποιούνται πολύ συχνά από τους προγραμματιστές των εφαρμογών MPI για να προσδιορίσουν την

προέλευση και τον προορισμό των μηνυμάτων. Επίσης πολύ συχνά χρησιμοποιούνται για τον έλεγχο της εκτέλεσης του προγράμματος (if rank=0 do this / if rank=1 do that).

Ομάδα (group): Ένα σύνολο N διατεταγμένων διεργασιών που χαρακτηρίζεται από τιμές τάξεων 0,1,2,…,N-1 είναι γνωστό ως ομάδα (group). Σε μια εφαρμογή MPI μπορούμε να ορίσουμε πολλές ομάδες διεργασιών κάθε μια από τις οποίες χαρακτηρίζεται από το δικό της κωδικό (group id).

Communicator: Ένα ακόμη σημαντικό χαρακτηριστικό του προτύπου MPI είναι η έννοια του μέσου επικοινωνίας (communicator), δια της χρήσεως του οποίου λαμβάνει χώρα η ανταλλαγή μηνυμάτων ανάμεσα σε δύο διεργασίες. Είναι προφανές πως για να είναι δυνατή η επικοινωνία ανάμεσα σε δύο διεργασίες αυτές θα πρέπει να ανήκουν στον ίδιο communicator. Η χρήση αυτού του τρόπου επικοινωνίας διασφαλίζει το σωστό τρόπο διακίνησης της πληροφορίας και ελαχιστοποιεί την πιθανότητα εμφάνισης προβλημάτων όσον αφορά την αποστολή και λήψη των μηνυμάτων από τις διεργασίες της εφαρμογής.

Ένας communicator ορίζεται ως ένα τοπικό αντικείμενο (local object) που χρησιμοποιείται για την αναπαράσταση ενός χώρου μέσα στον οποίο ανήκουν οι διεργασίες που έρχονται σε επικοινωνία (communication domain). Αυτός ο χώρος ορίζεται ως μια καθολική (global) και κατανεμημένη δομή που επιτρέπει στις διεργασίες μιας ομάδας να επικοινωνήσουν είτε μεταξύ τους, είτε με τις διεργασίες μιας άλλης ομάδας. Στην περίπτωση κατά την οποία οι διεργασίες που επικοινωνούν ανήκουν στην ίδια ομάδα, ο communicator είναι γνωστός ως intracommunicator ενώ στην περίπτωση κατά την οποία οι διεργασίες που επικοινωνούν ανήκουν σε διαφορετικές ομάδες, ο communicator είναι γνωστός ως intercommunicator.



Σχήμα 9. *Κάθε communicator συνδέει μια ομάδα διεργασιών*

3.3. Είδη επικοινωνιών ανάμεσα σε διεργασίες

3.3.1. Point-to-Point επικοινωνία

Ο βασικός μηχανισμός επικοινωνίας του MPI είναι η μετάδοση δεδομένων μεταξύ ενός ζεύγους διεργασιών, εκ των οποίων η πρώτη αποστέλλει το μήνυμα ενώ η δεύτερη το παραλαμβάνει. Αυτός ο τρόπος επικοινωνίας ονομάζεται επικοινωνία από σημείο σε σημείο (point-to-point communication) και αποτελεί το μοντέλο επικοινωνίας για ένα πολύ μεγάλο εύρος εφαρμογών MPI.
Υπάρχουν διάφορα είδη διαδικασιών επικοινωνίας από σημείο σε σημείο, τα οποία θα αναφερθούν στη συνέχεια.

Διαδικασίες επικοινωνίας point-to-point

Στην περίπτωση των διεργασιών από σημείο σε σημείο, οι συναρτήσεις εμφανίζονται σε δύο διαφορετικές μορφές που φέρουν τα ονόματα παρεμποδιστικές (blocking) και μη παρεμποδιστικές (non blocking) συναρτήσεις.

- **Παρεμποδιστική συνάρτηση:** Μια συνάρτηση λέγεται παρεμποδιστική όταν η κλήση της μέσα από κάποια διεργασία αναστέλλει τη λειτουργία αυτής της διεργασίας, μέχρι την ολοκλήρωση της λειτουργίας της συνάρτησης που έχει κληθεί.

- **Μη παρεμποδιστική συνάρτηση:** Μια συνάρτηση λέγεται μη παρεμποδιστική όταν η κλήση της δεν προκαλεί την αναστολή της λειτουργίας της διεργασίας.

Λαμβάνοντας υπόψη τη χρονική στιγμή περάτωσης της αποστολής του μηνύματος σε σχέση με τη χρονική στιγμή εκκίνησης της διαδικασίας παραλαβής του, μπορούμε να χωρίσουμε τις διαδικασίες επικοινωνίας από σημείο σε σημείο σε τέσσερις διαφορετικές μορφές.

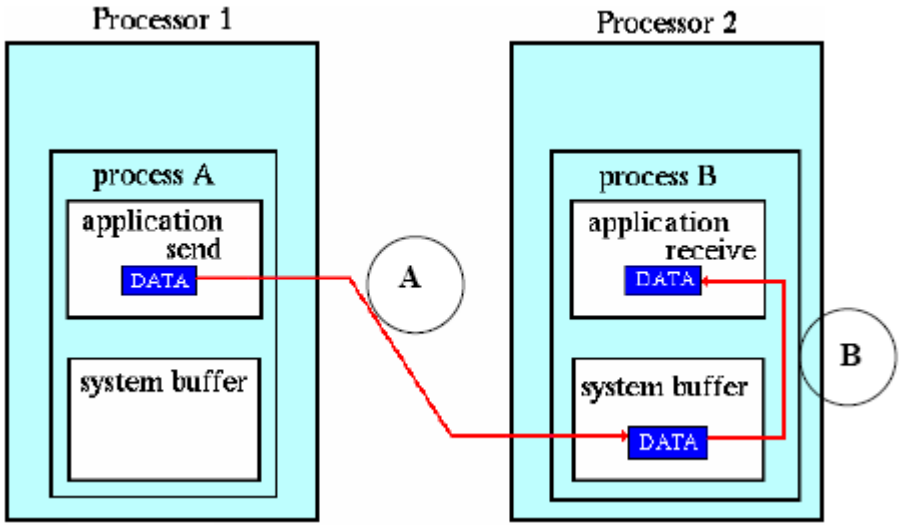
- **Πρότυπη επικοινωνία (standard communication):**
  Σε αυτόν τον τύπο επικοινωνίας η απόφαση για τη δέσμευση ή όχι ενδιάμεσης περιοχής μνήμης στην οποία θα αποθηκευτεί προσωρινά το εξερχόμενο μήνυμα, λαμβάνεται από το MPI και όχι από το χρήστη. Στην περίπτωση κατά την οποία πραγματοποιηθεί η διαδικασία της ενδιάμεσης αποθήκευσης του μηνύματος, η διαδικασία αποστολής του είναι δυνατό να ολοκληρωθεί ακόμη και εάν η αντίστοιχη διαδικασία παραλαβής του μηνύματος δεν έχει ξεκινήσει ακόμη. Στην αντίθετη περίπτωση όμως, κατά την οποία δεν είναι διαθέσιμη η περιοχή ενδιάμεσης μνήμης, η διαδικασία αποστολής θα ολοκληρωθεί μόνο όταν έχει ξεκινήσει η αντίστοιχη διαδικασία παραλαβής και το μήνυμα έχει μεταφερθεί στην περιοχή αποθήκευσης της διεργασίας παραλήπτη.

- **Επικοινωνία με ενδιάμεση μνήμη (buffered communication):**
  Σε αυτόν τον τύπο επικοινωνίας η διαδικασία αποστολής μπορεί να ξεκινήσει ανεξάρτητα από το εάν έχει ξεκινήσει η αντίστοιχη

διαδικασία παραλαβής, ενώ μπορεί ακόμη και να ολοκληρωθεί ανεξάρτητα από την εκκίνηση ή όχι της τελευταίας διαδικασίας. Στην περίπτωση αυτή ωστόσο είναι πιθανή η απαίτηση της τοπικής αποθήκευσης του εξερχόμενου μηνύματος κάτι που σημαίνει πως η εφαρμογή θα πρέπει να μεριμνήσει για την αναζήτηση και δέσμευση του κατάλληλου αποθηκευτικού χώρου. Εάν η μνήμη που δεσμεύεται για το λόγο αυτό είναι ανεπαρκής όσον αφορά το μέγεθός της και δεν μπορεί να διατηρήσει το μήνυμα, εμφανίζεται σφάλμα. Αλλιώς, το μήνυμα θα μεταφερθεί με επιτυχία στη διεργασία παραλήπτη και πλέον μπορούμε να αποδεσμεύσουμε την τοπική περιοχή μνήμης που είχε δεσμευθεί για τη διακίνηση του μηνύματος.

- **Σύγχρονη επικοινωνία (synchronous communication):** Σε αυτόν τον τύπο επικοινωνίας μια διαδικασία αποστολής μπορεί να ξεκινήσει ανεξάρτητα από την εκκίνηση της διαδικασίας παραλαβής από τη διεργασία παραλήπτη. Ωστόσο στην περίπτωση αυτή η αποστολή θα ολοκληρωθεί με επιτυχία μόνο όταν η διαδικασία παραλαβής έχει ξεκινήσει. Στην περίπτωση αυτή είναι δυνατή η χρήση του αποθηκευτικού χώρου της διαδικασίας αποστολέα για την αποστολή του επόμενου μηνύματος.

- **Επικοινωνία σε κατάσταση ετοιμότητας (ready state):** Σε αυτόν τον τύπο επικοινωνίας η διαδικασία επικοινωνίας μπορεί να ξεκινήσει μόνο όταν έχει ξεκινήσει και η αντίστοιχη διαδικασία παραλαβής. Στην αντίθετη περίπτωση η επικοινωνία χαρακτηρίζεται από την εμφάνιση σφαλμάτων και από εξαιρετικά χαμηλό βαθμό αξιοπιστίας.



Σχήμα 10. *Δέσμευση ενδιάμεσης μνήμης (buffer) για τον συγχρονισμό των διεργασιών*

Για κάθε μια από τις παραπάνω μορφές επικοινωνίας υπάρχουν ξεχωριστές συναρτήσεις ειδικά σχεδιασμένες για αυτό το σκοπό, οι οποίες μπορούν να χρησιμοποιηθούν τόσο στην παρεμποδιστική όσο και στη μη παρεμποδιστική τους μορφή. Σε αντίθεση με τη διαδικασία αποστολής, η παραλαβή ενός μηνύματος πραγματοποιείται πάντα από την ίδια συνάρτηση η οποία λειτουργεί με βάση τις αρχές της πρότυπης επικοινωνίας (standard communication).

3.3.2. Συλλογικές επικοινωνίες

Το δεύτερο είδος επικοινωνίας που υποστηρίζει το πρωτόκολλο MPI, είναι οι συλλογικές επικοινωνίες (collective communications) οι οποίες χαρακτηρίζονται από την ύπαρξη περισσότερων από δύο διαδικασιών.
Οι πιο σημαντικές από αυτές τις μορφές επικοινωνίας, είναι οι εξής:

- **Εκπομπή δεδομένων (broadcasting)**, όπου μια διεργασία αποστέλλει ένα μήνυμα σε όλες τις διαθέσιμες διεργασίες.

- **Συλλογή δεδομένων (gather),** όπου μια διεργασία συλλέγει τα δεδομένα που έχουν αποσταλεί από όλες τις διαθέσιμες διεργασίες.

- **Διασπορά δεδομένων (scattering),** όπου τα δεδομένα ενός μηνύματος διαμοιράζονται σε όλες τις υπόλοιπες διεργασίες.

- **Μείωση (reduce)**, όπου μια διεργασία συλλέγει δεδομένα από τις υπόλοιπες διεργασίες της ομάδας, και ταυτόχρονα υπολογίζει κάποια συνάρτηση αυτών των δεδομένων.

Σε όλες τις παραπάνω διεργασίες, υπάρχει μια κεντρική διεργασία, η οποία είτε αποστέλλει τις πληροφορίες στις υπόλοιπες διεργασίας της ομάδας, είτε συλλέγει τα μηνύματα που αποστέλλονται από αυτές. Αυτή η διεργασία ονομάζεται διεργασία ρίζα (root process).

3.4. Οι τύποι δεδομένων του MPI

Αν και οι συναρτήσεις του προτύπου MPI μπορούν να κληθούν μέσα από ένα πρόγραμμα γραμμένο στη γλώσσα προγραμματισμού C, οι τύποι δεδομένων των ορισμάτων των εν λόγω συναρτήσεων, δεν είναι αυτοί που χρησιμοποιούνται στη γλώσσα C, καθώς το MPI χρησιμοποιεί τους δικούς του τύπους δεδομένων. Ο επόμενος πίνακας (πίνακας 1) περιέχει τους στοιχειώδεις τύπους δεδομένων που χρησιμοποιούνται από τις συναρτήσεις του MPI και τους τύπους δεδομένων της γλώσσας C που αντιστοιχούν στους τύπους δεδομένων του MPI.

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long_double

Πίνακας 1. *Τύποι δεδομένων που υποστηρίζονται από το MPI*

3.5. Οι βασικές συναρτήσεις του MPI

Για την διαχείριση του περιβάλλοντος, το MPI χρησιμοποιεί διάφορες συναρτήσεις που εκτελούν συγκεκριμένες λειτουργίες, όπως για παράδειγμα η αρχικοποίηση και ο τερματισμός του περιβάλλοντος του MPI. Οι βασικές συναρτήσεις που χρησιμοποιούνται περιγράφονται παρακάτω:

- **MPI_Init**: Αρχικοποιεί το περιβάλλον MPI και δημιουργεί όλες τις δομές δεδομένων που είναι αναγκαίες για τη λειτουργία του. Η συνάρτηση αυτή θα πρέπει να κληθεί μία και μοναδική φορά, πριν την κλήση οποιασδήποτε άλλης συνάρτησης MPI.
  Το πρωτότυπο της συνάρτησης MPI_Init έχει τη μορφή MPI_Init (int * argc, char **arcv[]) όπου argc και argv είναι τα ορίσματα που χρησιμοποιούμε κατά την κλήση της συνάρτησης main.
  Το πρώτο όρισμα αναφέρεται στο πλήθος των ορισμάτων με τα οποία καλείται το εκτελέσιμο αρχείο της εφαρμογής, ενώ το δεύτερο είναι ένας πίνακας συμβολοσειρών που περιέχει τα ονόματα αυτών των ορισμάτων.

- **MPI_Finalize**: Τερματίζει τη λειτουργία του περιβάλλοντος του MPI και απελευθερώνει τη μνήμη που δεσμεύεται από τις δομές δεδομένων του προτύπου. Η συνάρτηση αυτή θα πρέπει να καλείται πάντοτε τελευταία μέσα από τον πηγαίο κώδικα της εφαρμογής. Η συνάρτηση MPI_Finalize καλείται χωρίς ορίσματα.

- **MPI_Comm_rank**: Επιστρέφει την τάξη της τρέχουσας διεργασίας. Η κλήση της συνάρτησης χαρακτηρίζεται από τη σύνταξη MPI_Comm_rank (MPI_comm comm, int * rank) όπου η μεταβλητή comm χρησιμοποιείται για τον καθορισμό του Communicator στον οποίο ανήκει η διεργασία, ενώ η μεταβλητή rank είναι μια ακέραια μεταβλητή που μετά την επιτυχή κλήση της συνάρτησης θα περιέχει την τάξη της τρέχουσας διεργασίας.

- **MPI_Comm_size**: Επιστρέφει το πλήθος των διεργασιών που περιλαμβάνονται στον τρέχοντα Communicator.
  Η κλήση αυτής της συνάρτησης χαρακτηρίζεται από τη σύνταξη MPI_Comm_size (MPI_comm comm, int * size) όπου η μεταβλητή comm χρησιμοποιείται για τον καθορισμό του Communicator του οποίου το μέγεθος θέλουμε να ανακτήσουμε, ενώ η μεταβλητή size είναι μια ακέραια μεταβλητή που μετά την επιτυχή κλήση της συνάρτησης θα περιέχει το πλήθος των διεργασιών που περιλαμβάνονται στον τρέχοντα Communicator.

- **MPI_Abort**: Τερματίζει όλες τις διεργασίες που περιλαμβάνονται στον τρέχοντα Communicator. Γενικά, η συνάρτηση MPI_Abort θα πρέπει να καλείται κάθε φορά που η MPI_Init αποτύχει να διαμορφώσει με επιτυχία το περιβάλλον του MPI.

- **MPI_Initialized**: Υποδεικνύει εάν η συνάρτηση MPI_Init έχει κληθεί και επιστρέφει τις λογικές τιμές 1 (αληθές) ή 0 (ψευδές). Όπως έχει αναφερθεί παραπάνω, η συνάρτηση MPI_Init πρέπει να κληθεί μια και μοναδική φορά σε ένα πρόγραμμα. Με τη χρήση της συνάτησης MPI_Initialized μπορούμε να διαπιστώσουμε εάν η MPI_Init έχει κληθεί προηγουμένως και να αποφύγουμε τυχόν προβλήματα που μπορεί να προκύψουν απο την κλήση της στην περίπτωση που το περιβάλλον του MPI είναι ήδη αρχικοποιημένο.

3.5.1. Παρεμποδιστικές συναρτήσεις

Όπως έχει ήδη αναφερθεί, παρεμποδιστικές είναι οι συναρτήσεις οι οποίες κατά το χρονικό διάστημα της λειτουργίας τους, αναστέλλουν τη λειτουργία της διεργασίας που τις καλεί. Οι κυριότερες παρεμποδιστικές συναρτήσεις ειναι οι εξής:

- **MPI_Send:** Χρησιμοποιείται για την αποστολή ενός μηνύματος σε κάποια από τις διεργασίες που περιλαμβάνονται στον τρέχοντα Communicator.
  Το πρωτότυπο της συνάρτησης MPI_Send έχει τη μορφή MPI_Send (void * buf, int count, MPI_Datatype dataType, int destRank, int mesageTag, MPI_Comm comm).
  Τα ορίσματα της συνάρτησης αφορούν τα εξής χαρακτηριστικά:
  - το όρισμα *buf* περιέχει τη διεύθυνση της περιοχής μνήμης στην οποία βρίσκονται τα δεδομένα προς αποστολή.
  - το όρισμα *count* περιέχει το πλήθος των στοιχείων προς αποστολή.

- το όρισμα *dataType* περιέχει τον τύπο δεδομένων για αυτά τα στοιχεία.
- το όρισμα *destRank* περιέχει την τάξη της διεργασίας προς την οποία πρόκειται να αποσταλεί το μήνυμα (διεργασία παραλήπτης).
- το όρισμα *messageTag* περιέχει την ετικέτα του μηνύματος που μας επιτρέπει να ξεχωρίσουμε αυτό το μήνυμα από τα υπόλοιπα. Εάν δεν θέλουμε να καθορίσουμε κάποια ετικέτα μπορούμε να καταχωρήσουμε σε αυτό το όρισμα την τιμή MPI_ANY_TAG.
- το όρισμα *comm* περιέχει τον Communicator στον οποίο ανήκουν οι διεργασίες αποστολέας και παραλήπτης.

- **MPI_Recv:** Χρησιμοποιείται για την παραλαβή ενός μηνύματος από κάποια από τις διεργασίες που ανήκουν στον τρέχοντα Communicator.
  Το πρωτότυπο της συνάρτησης MPI_Recv έχει τη μορφή MPI_Recv (void * buf, int count, MPI_Datatype dataType, int sourceRank, int messageTag, MPI_Comm comm, MPI_Status status).
  Η εν λόγω συνάρτηση χρησιμοποιείται με τον ίδιο σχεδόν τρόπο όπως η MPI_Send, με τη διαφορά ότι το τέταρτο όρισμα αναφέρεται στην τάξη της διεργασίας αποστολέα, ενώ υπάρχει και ένα επιπλέον όρισμα που δεν εμφανίζεται στη συνάρτηση MPI_Send.
  - το όρισμα *sourceRank* περιέχει την τάξη της διεργασίας αποστολέα.
  - το όρισμα *status* επιτρέπει την ανάκτηση πληροφοριών σχετικά με την κατάσταση παραλαβής του μηνύματος.

- **MPI_Wait:** Εάν μια διαδικασία αποστολής ή παραλαβής δεν έχει ακόμη ολοκληρωθεί, η συνάρτηση MPI_Wait αναμένει την ολοκλήρωσή της, αναστέλλοντας τη λειτουργία της διεργασίας μέσα από την οποία έχει κληθεί. Στην περίπτωση που λαμβάνουν χώρα πολλαπλές διαδικασίες, υπάρχει η δυνατότητα χρήσης κάποιας από τις συναρτήσεις MPI_Waitany, MPI_Waitall, MPI_Waitsome έτσι ώστε να καθοριστεί αν θα πρέπει η διεργασία να αναμένει να ολοκληρωθούν όλες ή μερικές απο αυτές.
  Η κλήση της συνάρτησης MPI_Wait ακολουθεί τη σύνταξη int MPI_Wait (MPI_Request * request, MPI_Status * status).
  - η τιμή του ορίσματος *request* είναι εκείνη που επιστρέφεται από τις μη παρεμποδιστικές συναρτήσεις MPI_Isend ή MPI_Irecv.
  - το όρισμα *status* θα περιέχει μετά την επιστροφή της συνάρτησης, τη ζητούμενη πληροφορία.

3.5.2. Μη παρεμποδιστικες συναρτήσεις

Οι μη παρεμποδιστικές συναρτήσεις αποστολής και λήψης δεδομένων, σε αντίθεση με τις παρεμποδιστικές, δεν αναστέλλουν τη λειτουργία της διεργασίας που τις καλεί, αλλά επιτρέπουν τη συνέχιση της εκτέλεσης τους ταυτόχρονα με την αποστολή ή την παραλαβή του μηνύματος.

Το βασικό χαρακτηριστικό των μη παρεμποδιστικών διαδικασιών, είναι η χρήση ειδικών αντικειμένων (request objects) για την ταυτοποίηση των λειτουργιών που πραγματοποιούνται και τη συσχέτιση της διαδικασίας αποστολής του μηνύματος με τη διαδικασία ολοκλήρωσής της. Τα αντικείμενα αυτά δημιουργούνται από το MPI κατά την εκκίνηση της λειτουργίας του, διατηρούνται στη μνήμη του συστήματος, και είναι προσπελάσιμα μόνο με τη χρήση ειδικών συναρτήσεων, οι οποίες επιστρέφουν ένα χειριστή (handle) προς αυτά.

Οι κυριότερες συναρτήσεις αυτής της κατηγορίας φέρουν τα ονόματα MPI_Isend και MPI_Irecv και είναι οι μη παρεμποδιστικές εκδόσεις των παρεμποδιστικών συναρτήσεων MPI_Send και MPI_Recv.

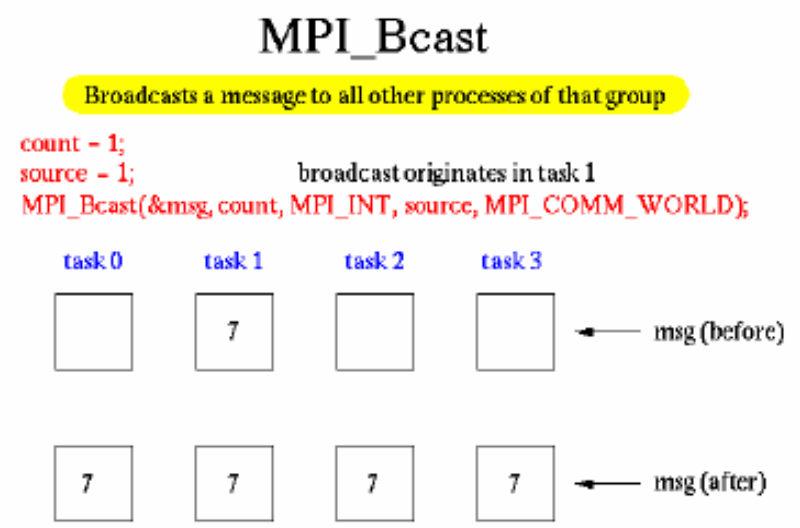
- **MPI_Isend:** Λειτουργεί με παρόμοιο τρόπο με τη συνάρτηση MPI_Send, καθώς και αυτή η συνάρτηση χρησιμοποιείται για την αποστολή ενός μηνύματος.
  Το πρωτότυπο της συνάρτησης MPI_Isend έχει τη μορφή int MPI_Isend (void * buf, int count, MPI_Datatype dataType, int destRank, int messageTag, MPI_Comm comm, MPI_Request request).
  Παρατηρούμε πως η μόνη διαφορά σε σχέση με τη σύνταξη της συνάρτησης MPI_Send, είναι το τελευταίο όρισμα της συνάρτησης MPI_Isend, το οποίο δεν εμφανίζεται στην MPI_Send. Το όρισμα αυτό (request) είναι μία μεταβλητή τύπου MPI_Request η οποία αποτελεί χειριστή του αδιαφανούς αντικειμένου που χρησιμοποιείται για την πραγματοποίηση της επικοινωνίας.

- **MPI_Irecv:** Χρησιμοποιείται για τη μη παρεμποδιστική διαδικασία παραλαβής μηνύματος, επομένως λειτουργεί και αυτή με παρόμοιο τρόπο με τη συνάρτηση MPI_Recv.
  Το πρωτότυπο της συνάρτησης MPI_Irecv έχει τη μορφή int MPI_Irecv (void * buf, int count, MPI_Datatype dataType, int srcRank, int messageTag, MPI_Comm comm, MPI_Request request).
  Η σημασία των ορισμάτων είναι όμοια με τη συνάρτηση MPI_Isend.
  Παρατηρούμε πως η συνάρτηση MPI_Irecv δεν περιέχει όρισμα τύπου MPI_Status, όπως η συνάρτηση MPI_Recv. Το γεγονός αυτό συμβαίνει γιατί η μεταβλητή status στη συνάρτηση MPI_Recv περιέχει πληροφορίες σχετικά με την διαδικασία παραλαβής και επομένως δεν μπορεί να χρησιμοποιηθεί στη συνάρτηση MPI_Irecv, η οποία απλώς εκκινεί τη διαδικασία παραλαβής του μηνύματος και στη συνέχεια τερματίζει τη λειτουργία της.

3.5.3. Συναρτήσεις συλλογικής επικοινωνίας

Οι βασικότερες συναρτήσεις που επιτρέπουν την πραγματοποίηση συλλογικών επικοινωνιών είναι:

- **MPI_Bcast:** Αποστέλλει ένα μήνυμα από τη διεργασία ρίζα σε όλες τις υπόλοιπες διεργασίες της τρέχουσας ομάδας (σχήμα 13).
  Η χρήση της συνάρτησης MPI_Bcast χαρακτηρίζεται από μία σύνταξη της μορφής int MPI_Bcast (void * buffer, int count, MPI_Datatype dataType, int root, MPI_Comm comm).
  - το όρισμα *buffer* αναφέρεται στην περιοχή μνήμης που περιέχει το εν λόγω μήνυμα.
  - το όρισμα *count* καθορίζει το πλήθος των στοιχείων του μηνύματος.
  - το όρισμα *dataType* καθορίζει τον τύπο δεδομένων των στοιχείων του μηνύματος.
  - το όρισμα *root* περιέχει την τάξη της διεργασίας που θα αναλάβει την αποστολή του μηνύματος.
  - το όρισμα *comm* αναφέρεται κατά τα γνωστά στον communicator που χρησιμοποιείται για τη διακίνηση των μηνυμάτων.

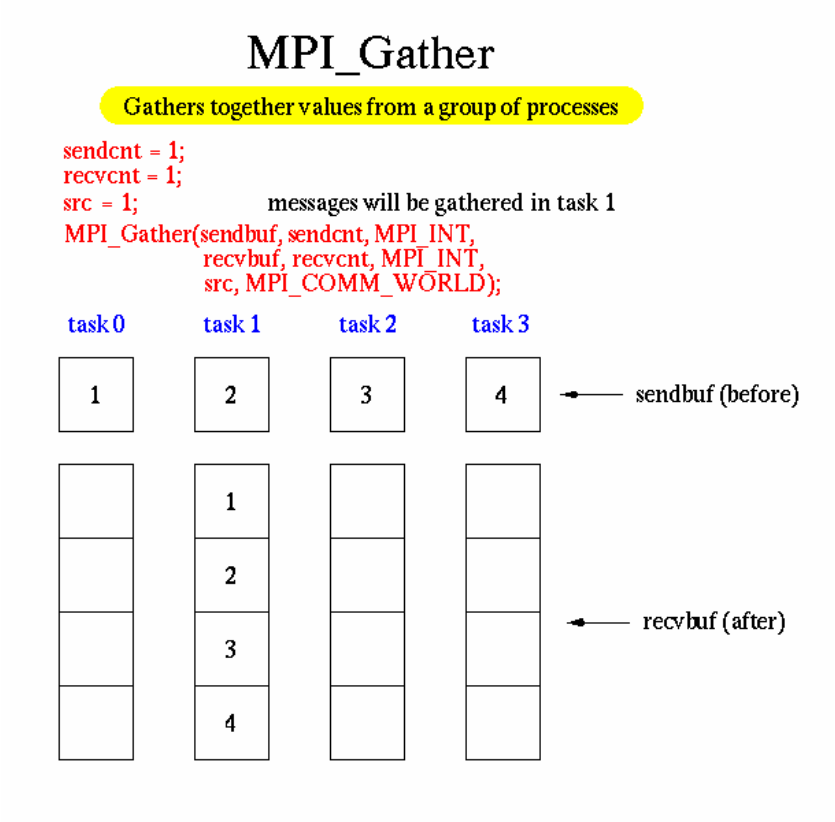


Σχήμα 13. *Παρουσίαση της λειτουργίας της συνάρτησης MPI_Bcast*

- **MPI_Gather:** Επιτρέπει σε μια διεργασία να συλλέξει δεδομένα που έχουν αποσταλεί προς αυτή από όλες τις υπόλοιπες διεργασίες της τρέχουσας ομάδας (σχήμα 14).
  Η MPI_Gather καλείται τόσο από τις διεργασίες που αποστέλλουν τα δεδομένα όσο και από τη διεργασία ρίζα η οποία θα πραγματοποιήσει την παραλαβή των δεδομένων. Σε αυτόν τον τρόπο επικοινωνίας, η κάθε διεργασία (συμπεριλαμβανομένης και της διεργασίας ρίζα) αποστέλλει τα δεδομένα της στη διεργασία ρίζα η

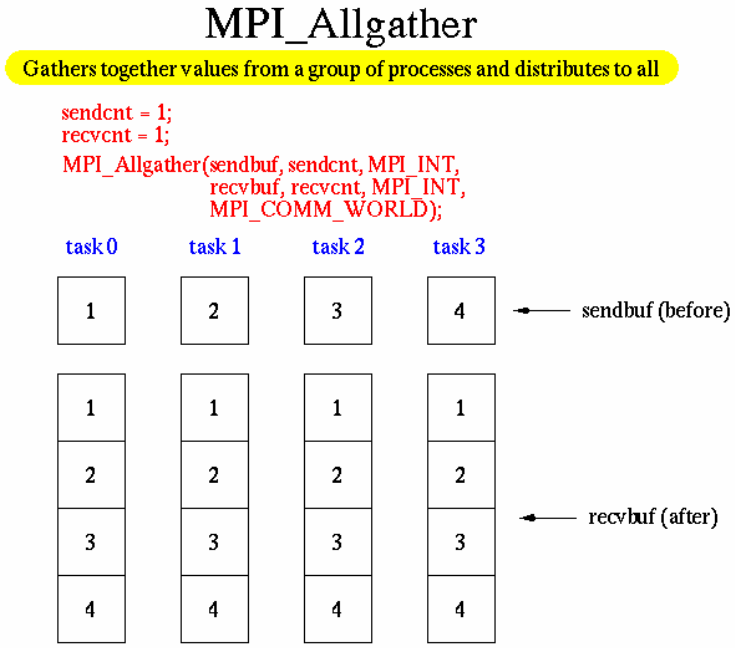
οποία τα παραλαμβάνει και τα αποθηκεύει, ανάλογα με την τιμή της τάξης της διεργασίας που τα απέστειλε.
Η χρήση της συνάρτησης MPI_Gather ακολουθεί τη σύνταξη int MPI_Gather (void * sendBuf, int sendCount, MPI_Datatype sendType, void * recvBuf, int recvCount, MPI_Datatype recvType, int root, MPI_Comm comm).

- το όρισμα *sendBuf* αναφέρεται στην περιοχή μνήμης της διαδικασίας αποστολέα.
- το όρισμα *sendCount* καθορίζει το πλήθος των στοιχείων προς αποστολή.
- το όρισμα *sendType* καθορίζει τον τύπο δεδομένων των στοιχείων προς αποστολή.
- το όρισμα *recvBuf* περιγράφει την περιοχή μνήμης στην οποία θα αποθηκευτούν αυτά τα δεδομένα από τη διεργασία ρίζα.
- το όρισμα *recvCount* καθορίζει το πλήθος των στοιχείων που θα παραληφθούν από κάθε διεργασία.
- το όρισμα *recvType* καθορίζει τον τύπο δεδομένων των στοιχείων που θα παραληφθούν από κάθε διεργασία.
- το όρισμα *root* καθορίζει την τιμή της τάξης της διεργασίας ρίζα.
- το όρισμα *comm* αναφέρεται ως συνήθως στον communicator που χρησιμοποιείται για τη διακίνηση των δεδομένων ανάμεσα στις διεργασίες του συστήματος.



Σχήμα 14. *Παρουσίαση λειτουργίας της συνάρτησης MPI_Gather.*

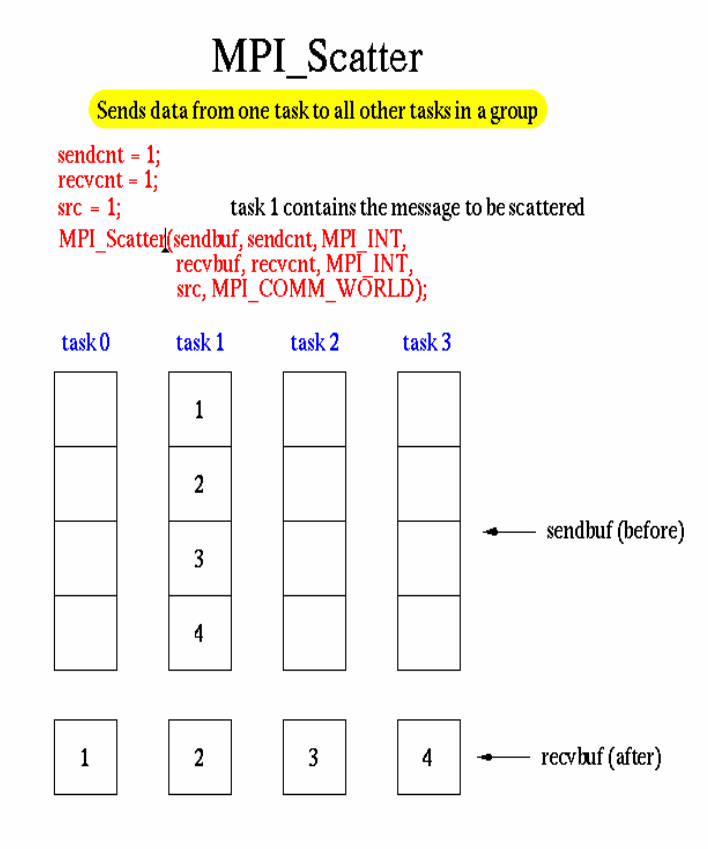
- **MPI_Allgather:** Αποτελεί γενίκευση της MPI_Gather. Επιτρέπει την παραλαβή του αποτελέσματος, όχι μόνο από τη διεργασία ρίζα αλλά και από όλες τις διεργασίες της τρέχουσας ομάδας (σχήμα 15).
  Η χρήση της συνάρτησης MPI_Allgather χαρακτηρίζεται από μία σύνταξη της μορφής int MPI_Allgather (void * sendBuf, int sendCount, MPI_Datatype sendType, void * recvBuf, int recvCount, MPI_Datatype recvType, MPI_Comm comm).
  Η σημασία των ορισμάτων της συνάρτησης MPI_Allgather είναι ίδια με εκείνη της συνάρτησης MPI_Gather. Παρατηρούμε ωστόσο πως στην περίπτωση αυτή δεν υπάρχει το όρισμα root που αναφέρεται στη διεργασία ρίζα, κάτι που είναι αναμενόμενο, διότι τα δεδομένα του συστήματος παραλαμβάνονται από όλες τις διαθέσιμες διεργασίες.



Σχήμα 15. *Παρουσίαση λειτουργίας της συνάρτησης MPI_Allgather*

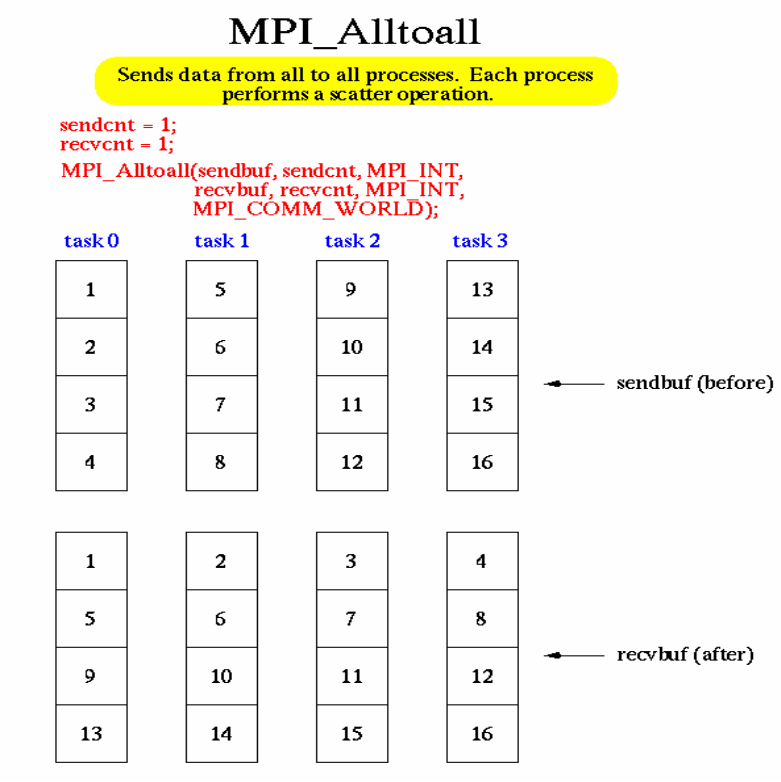
- **MPI_Scatter:** Επιτρέπει τη διασπορά των δεδομένων ενός μηνύματος σε όλες τις διεργασίες της εφαρμογής (σχήμα 16).
  Η διασπορά, ως λειτουργία, θεωρείται η αντίστροφη της συλλογής και χαρακτηρίζεται από το διαχωρισμό ενός μηνύματος σε μικρότερα κομμάτια, κάθε ένα εκ των οποίων αποστέλλεται και σε μια διαφορετική διεργασία.
  Η χρήση της συνάρτησης MPI_Scatter χαρακτηρίζεται από μία σύνταξη τηςμορφής int MPI_Scatter (void * sendBuf, int sendCount, MPI_Datatype sendType, void * recvBuf, int recvCount, MPI_Datatype recvType, int root, MPI_Comm comm).
  Τα ορίσματα στην παραπάνω σύνταξη της MPI_Scatter έχουν την ίδια σημασία με τα αντίστοιχα ορίσματα της συνάρτησης MPI_Gather που περιγράφτηκαν παραπάνω.

- το όρισμα *sendBuf* αναφέρεται στην περιοχή μνήμης στην οποία αποθηκεύεται το μήνυμα προς διασπορά.
- το όρισμα *sendCount* καθορίζει το πλήθος των στοιχείων του μηνύματος.
- το όρισμα *sendType* καθορίζει τον τύπο δεδομένων των στοιχείων του μηνύματος.
- το όρισμα *recvBuf* περιγράφει την περιοχή μνήμης στην οποία θα αποθηκευτούν αυτά τα δεδομένα από τη διεργασία παραλήπτη.
- το όρισμα *recvCount* καθορίζει το πλήθος των στοιχείων που παραλαμβάνονται από κάθε διεργασία.
- το όρισμα *recvType* καθορίζει τον τύπο δεδομένων των στοιχείων παραλαμβάνονται από κάθε διεργασία.
- το όρισμα *root* ταυτοποιεί τη διεργασία ρίζα που πρόκειται να αποστείλει το μήνυμα.
- το όρισμα *comm* ορίζει τον communicator που χρησιμοποιείται για τη διακίνηση των δεδομένων ανάμεσα στις διεργασίες του συστήματος.



Σχήμα 16. *Παρουσίαση λειτουργίας της συνάρτησης MPI_Scatter*

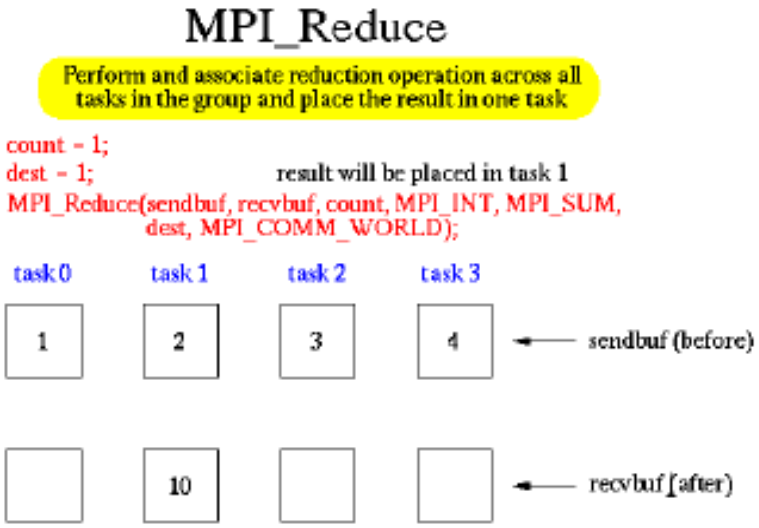
- **MPI_Alltoall:** Επιτρέπει τη συνδυασμένη χρήση των συναρτήσεων αποστολής και παραλαβής δεδομένων. Πιο συγκεκριμένα, η συνάρτηση αυτή αποτελεί μια επέκταση της MPI_Allgather και επιτρέπει την αποστολή δεδομένων από κάθε διεργασία του συστήματος σε όλες τις διεργασίες της τρέχουσας ομάδας, οι οποίες και παραλαμβάνουν αυτά τα δεδομένα (σχήμα 17).
  Η χρήση της συνάρτησης MPI_Alltoall χαρακτηρίζεται από μία σύνταξη της μορφής int MPI_Alltoall (void * sendBuf, int sendCount, MPI_Datatype sendType, void * recvBuf, int recvCount, MPI_Datatype recvType, MPI_Comm comm).
  Η σημασία των ορισμάτων είναι η ίδια με εκείνη που παρουσιάσαμε κατά την περιγραφή των προηγούμενων συλλογικών συναρτήσεων.



Σχήμα 17. *Παρουσίαση λειτουργίας της συνάρτησης MPI_Alltoall*

- **MPI_Reduce:** Δέχεται ως είσοδο ένα πίνακα τιμών που έχουν αποσταλεί από άλλες διεργασίες και υπολογίζει την τιμή μιας απλής ποσότητας, εφαρμόζοντας πάνω στις τιμές του διανύσματος εισόδου μια αριθμητική ή λογική πράξη (σχήμα 18).
  Η χρήση της συνάρτησης MPI_Reduce χαρακτηρίζεται από μια σύνταξη της μορφής int MPI_Reduce (void * sendBuffer, void * recvBuffer, int count, MPI_Datatype dataType, MPI_Op operation, int root, MPI_Comm comm).

- το όρισμα *sendBuffer* αναφέρεται στην περιοχή μνήμης που περιέχει τα δεδομένα εισόδου της συνάρτησης.
- το όρισμα *recvBuffer* περιγράφει την περιοχή μνήμης στην οποία θα αποθηκευτεί το αποτέλεσμα της πράξης που εφαρμόζεται.
- το όρισμα *count* καθορίζει το πλήθος των στοιχείων προς αποστολή.
- το όρισμα *dataType* καθορίζει τον τύπο δεδομένων των στοιχείων προς αποστολή.
- το όρισμα *operation* καθορίζει το είδος της πράξης που θα εφαρμοστεί.
- το όρισμα περιγράφει την τάξη της διεργασίας ρίζα που θα πραγματοποιήσει την πράξη της αναγωγής.
- το όρισμα *comm* ορίζει τον communicator που χρησιμοποιείται για τη διακίνηση των δεδομένων της εφαρμογής.



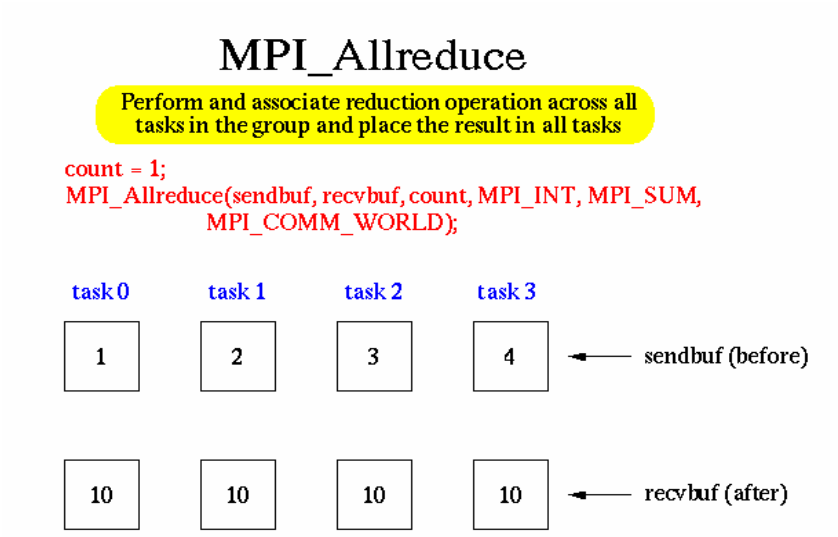
Σχήμα 18. *Παρουσίαση λειτουργίας της συνάρτησης MPI_Reduce*

Το πρότυπο MPI παρέχει τη δυνατότητα χρήσης ενός αρκετά μεγάλου αριθμού αριθμητικών και λογικών πράξεων που μπορούν να εφαρμοστούν στα δεδομένα εισόδου, αν και ο χρήστης έχει τη δυνατότητα να ορίσει και αυτός τις δικές του πράξεις. Ο πίνακας που ακολουθεί (πίνακας 2) παρουσιάζει τις τιμές του ορίσματος operation και την πράξη που αντιστοιχεί σε κάθε μια από αυτές τις τιμές.

Τιμή ορίσματος Operation	Αριθμητική ή λογική πράξη
MPI_MIN	Επιστρέφει το στοιχείο με τη μικρότερη τιμή
MPI_MAX	Επιστρέφει το στοιχείο με τη μεγαλύτερη τιμή
MPI_SUM	Επιστρέφει το άθροισμα των στοιχείων εισόδου
MPI_PROD	Επιστρέφει το γινόμενο των στοιχείων εισόδου
MPI_LAND	Εφαρμόζει την πράξη της λογικής σύζευξης
MPI_BAND	Εφαρμόζει την πράξη της δυαδικής σύζευξης
MPI_LOR	Εφαρμόζει την πράξη της λογικής διάζευξης
MPI_BOR	Εφαρμόζει την πράξη της δυαδικής διάζευξης
MPI_LXOR	Εφαρμόζει την πράξη της λογικής αποκλειστικής διάζευξης
MPI_BXOR	Εφαρμόζει την πράξη της δυαδικής αποκλειστικής διάζευξης
MPI_MAXLOC	Επιστρέφει τη θέση και την τιμή του μεγαλύτερου στοιχείου
MPI_MINLOC	Επιστρέφει τη θέση και την τιμή του μικρότερου στοιχείου

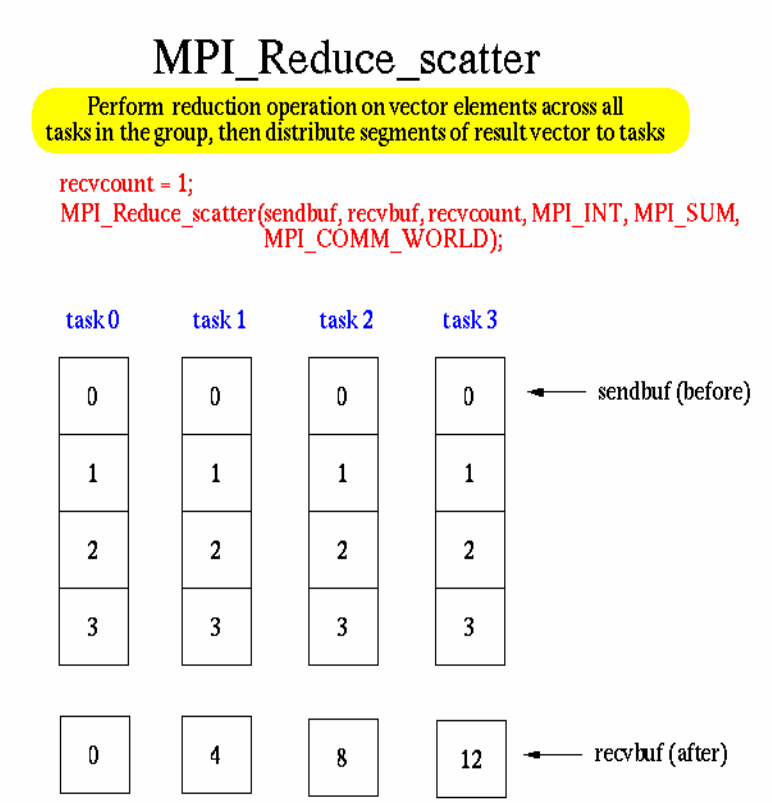
Πίνακας 2. *MPI Reduction operations*

- **MPI_Allreduce:** Αποτελεί γενίκευση της συνάρτησης MPI_Reduce. Αποστέλλει το αποτέλεσμα της αναγωγής που πραγματοποιείται από τη διεργασία ρίζα σε όλες τις διεργασίες της τρέχουσας ομάδας (σχήμα 19).
  Η χρήση της συνάρτησης MPI_Allreduce χαρακτηρίζεται από μία σύνταξη της μορφής int MPI_Allreduce (void * sendBuf, void * recvBuf, int count, MPI_Datatype dataType, MPI_Op operation, MPI_Comm comm).
  Τα ορίσματα της συνάρτησης έχουν την ίδια σημασία με τα αντίστοιχα ορίσματα της συνάρτησης MPI_Reduce που περιγράφτηκαν παραπάνω.



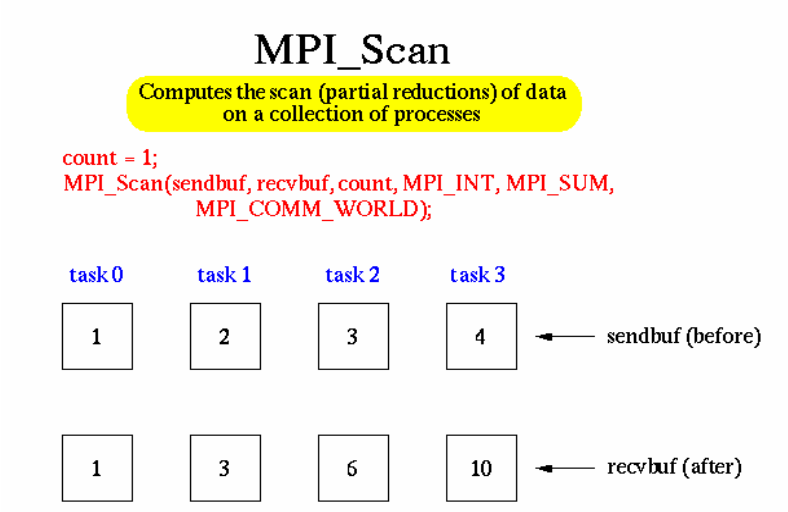
Σχήμα 19. *Παρουσίαση λειτουργίας της συνάρτησης MPI_Allreduce.*

- **MPI_Reduce_scatter:** Μετά τον υπολογισμό της εξόδου της πράξης της αναγωγής που πραγματοποιείται από τη διεργασία ρίζα, η συνάρτηση αυτή προχωρεί σε διασπορά του αποτελέσματος σε όλες τις διαθέσιμες διεργασίες (σχήμα 20).
  Η χρήση της συνάρτησης MPI_Reduce_scatter χαρακτηρίζεται από μία σύνταξη της μορφής int MPI_Reduce_scatter (void * sendBuf, void * recvBuf, int * recvCount, MPI_Datatype dataType, MPI_Op operation, MPI_Comm comm).
  Η σημασία των ορισμάτων είναι η ίδια με εκείνη που περιγράφτηκε κατά την παρουσίαση της συνάρτησης MPI_Reduce.



Σχήμα 20. *Παρουσίαση λειτουργίας της συνάρτησης MPI_Reduce_scatter*

- **MPI_Scan:** Πραγματοποιεί την αναγωγή ενός πλήθους στοιχείων ο αριθμός των οποίων εξαρτάται από την τάξη της κάθε διεργασίας. Πιο συγκεκριμένα, η συνάρτηση αυτή επιστρέφει στην περιοχή μνήμης παραλαβής της διεργασίας υπ' αριθμόν i το αποτέλεσμα της αναγωγής των στοιχείων που έχουν αποσταλεί από τις διεργασίες με τιμές τάξης 0,1,2…,i (σχήμα 21).
  Η χρήση της συνάρτησης MPI_Scan χαρακτηρίζεται από μια σύνταξη της μορφής int MPI_Scan (void * sendBuf, void * recvBuf, int count, MPI_Datatype dataType, MPI_Op operation, MPI_COMM_WORLD comm).
  Η σημασία των ορισμάτων είναι η ίδια με εκείνη που περιγράφτηκε κατά την παρουσίαση της συνάρτησης MPI_Reduce.



Σχήμα 21. *Παρουσίαση λειτουργίας της συνάρτησης MPI_Scan*

ΚΕΦΑΛΑΙΟ 4

4. Ο κανόνας του Simpson

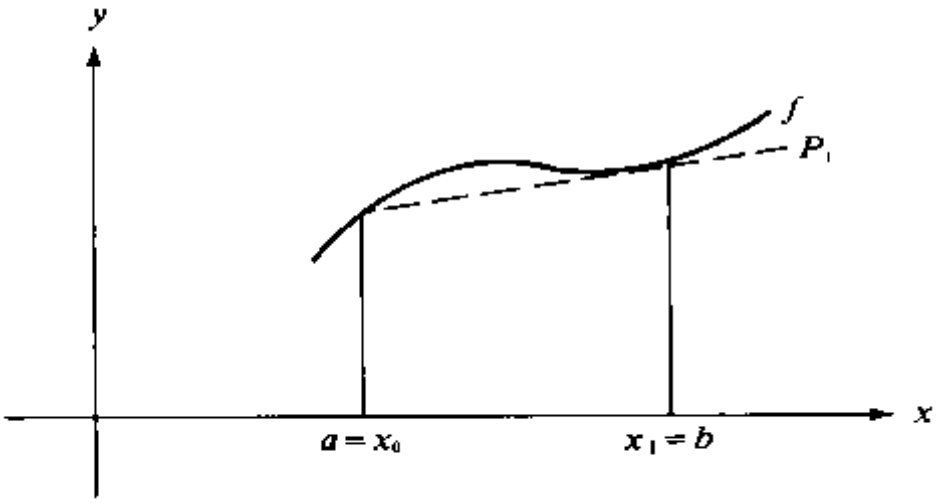
Συχνά απαιτείται ο υπολογισμός ολοκληρώματος συναρτήσεων των οποίων η αναλυτική μορφή του ολοκληρώματος δεν είναι γνωστή ή δεν μπορεί να υπολογιστεί. Η βασική μέθοδος που χρησιμοποιείται για την προσέγγιση της τιμής ενός ολοκληρώματος βασίζεται στη χρήση αθροίσματος της μορφής :

$$\int_a^b f(x)dx = \sum_{i=1}^{n} a_i f(x_i) \quad (4.1)$$

Σύμφωνα με τον **κανόνα του Τραπεζίου** έχουμε :

$$\int_a^b f(x)dx \cong \frac{h}{2}\left[f(x_0)+f(x_1)\right] \quad (4.2)$$

όπου x_0=a, x_1=b και h=b-a

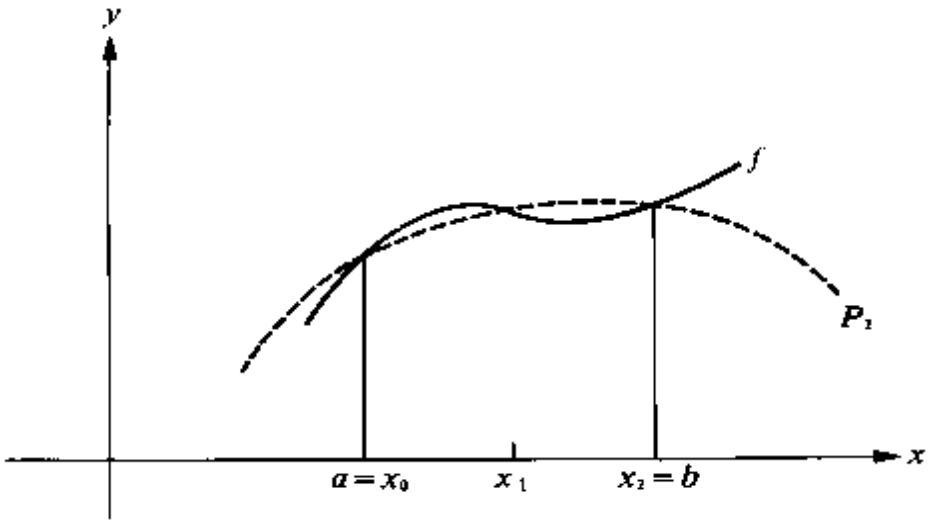


Σχήμα 22. *Εφαρμογή του κανόνα του Τραπεζίου σε αριθμητική ολοκλήρωση*

Σύμφωνα με τον **κανόνα του Simpson** έχουμε :

$$\int_a^b f(x)dx \cong \frac{h}{3}\left[f(x_0)+4f(x_1)+f(x_2)\right] \quad (4.3)$$

όπου x_0=a, x_1=a+h, x_2=b και h=(b-a)/2



Σχήμα 23. *Εφαρμογή του κανόνα του Simpson*

Για την υλοποίηση του κανόνα του Simpson σε MPI επιλέξαμε την συνάρτηση :

f(x)=4/(1+x^2)

Το πρόγραμμα που υλοποιήθηκε δέχεται τρία ορίσματα :

1. το όριο a (σχήμα 23)
2. το όριο b (σχήμα 23)
3. τον αριθμό των υπο-διαστημάτων στον οποίο θα διαχωριστεί το διάστημα [a, b]

Κατά την εκτέλεση του προγράμματος που υλοποιήθηκε ακολουθούνται τα παρακάτω βήματα :

1. Υπολογίζεται ο αριθμός των υπο-διαστημάτων που θα αναλάβει κάθε slave διεργασία/node να υπολογίσει.
2. Η κύρια διεργασία/master node στέλνει σε κάθε slave διεργασία/node τα a_i, b_i (που ανήκουν στο [a, b]), τα οποία αποτελούν τα όρια του διαστήματος για τα οποία η slave διεργασία/node θα πρέπει να υπολογίσει το εμβαδό καθώς επίσης και την τιμή του συνολικού αποτελέσματος τη συγκεκριμένη χρονική στιγμή.
3. Κάθε slave διεργασία/node αφού πάρει τα a_i, b_i και το τρέχον συνολικό αποτέλεσμα από την master διεργασία/node, υπολογίζει το εμβαδό για το συγκεκριμένο υπο-διάστημα, το προσθέτει στο συνολικό αποτέλεσμα και στη συνέχεια στέλνει τη νέα τιμή του συνολικού αποτελέσματος στην master διεργασία/node.
4. Όταν το εμβαδό όλων των υποδιαστημάτων έχει υπολογιστεί από τις slave διεργασίες/nodes, η master διεργασία/node επιστρέφει το συνολικό εμβαδό που ορίζεται από τη συνάρτηση f(x) για το διάστημα [a, b].

4.1. Υλοποίηση του κανόνα του Simpson σε MPI

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "mpi.h"

int subIntervals_counter;
int slave_calculations_counter;
int intervals_counter;
double border_a;
double border_b;
int subIntervals_number;
double interval_length;     //the length of each interval
double temp_x1;
double temp_x2;
double final_result;


double myPow(double myBase, int power)      //function that calculates and returns the power of a number
{
    int j;
    double result = 1;

    if(power == 0)
        result = 1;
    else
    {
        for(j=0; j<power; j++)
            result = result * myBase;
    }
    return result;
}

double f(double x)          //it returns the function's result for a given "x"
{
    double result = 0;      //result's initialization
    result = 4/(1+myPow(x,2));
    return result;
}

//function which calculates the area of the trapezium formed by joining the points (x1, 0), (x2, 0),
//(x1, f(x1)) and (x2, f(x2)).
double calculateArea(double a, double b)
{
    double result = 0;

    double h = (b-a)/2;
    double x0 = a;
    double x1 = a + h;
    double x2 = b;

    result = (h/3) * (f(x0) + 4*f(x1) + f(x2));

    return result;
}
```

```
int main(int argc, char* argv[])
{
    int calculations_for_each_slave; //stores the minimum number of calculations that a node will execute
    int calculations_remainder;      //stores the number of the nodes that will execute one more calculation

    int rank, size, i;
    MPI_Status status;

    border_a = atof(argv[1]);                       //get argv[1], which is the border a
    border_b = atof(argv[2]);                       //get argv[2], which is the border b
    subIntervals_number = atoi(argv[3]);            //get argv[3], which is the number of subintervals

    interval_length = (border_b - border_a)/subIntervals_number;  //calculate the length of each interval

    //initialization
    final_result = 0;
    intervals_counter = 0;
    temp_x1 = 0;
    temp_x2 = border_a;

    /* Initialize (start) MPI */
    MPI_Init(&argc, &argv);

    /* who am I? How many nodes are available? */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);


    if( size != 1)  //if there are more than one nodes
    {
        /*************************************************/
        /* we have size-1 slaves which will              */
        /* execute at minimum "calculations_for_each_slave" */
        /* calculations each                             */
        /*************************************************/
        calculations_for_each_slave = subIntervals_number/(size-1);
        /************************************************/
        /* the first "calculations_remainder" slave nodes */
        /* will execute one more calculation            */
        /************************************************/
        calculations_remainder = subIntervals_number%(size-1);

        i = 0;  //stores the ID of the current node

        /* I am the master node */
        if ( rank == 0 )
        {
            //send the boundaries for each interval to the slave nodes
            for(subIntervals_counter=1; subIntervals_counter<=subIntervals_number; subIntervals_counter++)
            {
                i++;
                if(i == size)       //when we reach the last slave node
                    i = 1;          //we start from the beginning
```

```
        temp_x1 = temp_x2;      //take the next interval
        temp_x2 = temp_x1 + interval_length;

        printf("\n  --- Master says : I send data for interval No.%d ---\n",subIntervals_counter);

        /* send temp_x1 */
        MPI_Send(&temp_x1,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
        printf("Master : temp_x1 = %g sent to slave node No:%d\n",temp_x1,i);
        /* send temp_x2 */
        MPI_Send(&temp_x2,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
        printf("Master : temp_x2 = %g sent to slave node No:%d\n",temp_x2,i);
        /* send final_result */
        MPI_Send(&final_result,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD);


        //receive final_result
        MPI_Recv(&final_result,1,MPI_DOUBLE,i,2,MPI_COMM_WORLD,&status);
        printf("Master : I RECEIVED final_result = %g from slave node No.%d\n",final_result,i);
    }
    printf("\n\n\n\n--- THE FINAL RESULT IS : %g ---\n\n\n\n",final_result);    //print the final result
}
/* I am a slave */
else
{
    //each slave node executes at minimum "calculations_for_each_slave" number of calculations
    for(slave_calculations_counter=0; slave_calculations_counter<calculations_for_each_slave; slave_calculations_counter++)
    {
        //receive temp_x1
        MPI_Recv(&temp_x1,1,MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);
        printf("slave node No.%d : I received temp_x1 = %g\n",rank,temp_x1);
        //receive temp_x2
        MPI_Recv(&temp_x2,1,MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);
        printf("slave node No.%d : I received temp_x2 = %g\n",rank,temp_x2);
        //receive final_result
        MPI_Recv(&final_result,1,MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);


        //send to master the updated result
        final_result = final_result + calculateArea(temp_x1, temp_x2);

        printf("slave node No.%d : I SEND TO MASTER final_result = %g\n",rank,final_result);
        MPI_Send(&final_result,1,MPI_DOUBLE,0,2,MPI_COMM_WORLD);
    }
    //then, some of the slave nodes execute the rest of the calculations
    for(slave_calculations_counter=1; slave_calculations_counter<=calculations_remainder; slave_calculations_counter++)
    {
        if(rank == slave_calculations_counter)
        {
            //receive temp_x1
            MPI_Recv(&temp_x1,1,MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);
            printf("slave node No.%d : I received temp_x1 = %g\n",rank,temp_x1);
            //receive temp_x2
            MPI_Recv(&temp_x2,1,MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);
            printf("slave node No.%d : I received temp_x2 = %g\n",rank,temp_x2);
            //receive final_result
            MPI_Recv(&final_result,1,MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);

                    //send to master the updated result
                    final_result = final_result + calculateArea(temp_x1, temp_x2);

                    printf("slave node No.%d : I SEND TO MASTER final_result = %g\n",rank,final_result);
                    MPI_Send(&final_result,1,MPI_DOUBLE,0,2,MPI_COMM_WORLD);
                }
            }
        }
    }
    else //there is only one node - the program DOES NOT run in parallel
    {
        for(subIntervals_counter=1; subIntervals_counter<=subIntervals_number; subIntervals_counter++)
        {
            temp_x1 = temp_x2;      //take the next interval
            temp_x2 = temp_x1 + interval_length;
            final_result = final_result + calculateArea(temp_x1, temp_x2);
        }
        printf("\n\n\n\n--- THE FINAL RESULT IS : %g ---\n\n\n\n",final_result);    //print the final result
    }

    /* stop MPI */
    MPI_Finalize();
    /* exit program without error (code 0)*/
    return(0);
}
```

4.1.1. Επεξήγηση του πηγαίου κώδικα

Στην αρχή του προγράμματος ορίζουμε κάποιες καθολικές μεταβλητές :

```
double border_a;
double border_b;
int subIntervals_number;
int subIntervals_counter;
double interval_length;
int slave_calculations_counter;
double temp_x1;
double temp_x2;
double final_result;
```

Στις μεταβλητές **border_a** και **border_b** αποθηκεύονται τα όρια του διαστήματος [a, b]. Η μεταβλητή **subIntervals_number** ορίζει τον αριθμό των υπο-διαστημάτων στα οποία θα χωριστεί το διάστημα [a, b]. Η μεταβλητή **subIntervals_counter** είναι ένας μετρητής των υπο-διαστημάτων αυτών. Το μήκος κάθε υπο-διαστήματος αποθηκεύεται στην μεταβλητή **interval_length.** Η μεταβλητή **slave_calculations_counter** δηλώνει τον αριθμό των υπολογισμών που θα εκτελέσει κάθε slave διεργασία. Στις μεταβλητές **temp_x1** και **temp_x2** αποθηκεύονται προσωρινά τα όρια του τρέχοντος υπο-διαστήματος. Τέλος, στη μεταβλητή **final_result** αποθηκεύται το τελικό αποτέλεσμα, δηλαδή η τιμή του συνολικού εμβαδού που ορίζεται από τη συνάρτηση f(x) για το διάστημα [a, b].

```
double myPow(double myBase, int power)
{
	int j;
	double result = 1;

	if(power == 0)
		result = 1;
	else
	{
		for(j=0; j<power; j++)
			result = result * myBase;
	}
	return result;
}
```

Η συνάρτηση **myPow** χρησιμοποιείται για τον υπολογισμό της δύναμης ενός αριθμού. Τα ορίσματα της συνάρτησης είναι η μεταβλητή **myBase** στην οποία αποθηκεύεται η βάση και η μεταβλητή **power** στην οποία αποθηκεύεται ο εκθέτης. Η μεταβλητή **j** αποτελεί έναν μετρητή ενώ στο **result** αποθηκεύεται το τελικό αποτέλεσμα.

Επειδή όταν ο εκθέτης είναι το μηδέν για οποιαδήποτε βάση το αποτέλεσμα ισούται με 1, έχουμε :

```
	if(power == 0)
		result = 1;
```

Σε αντίθετη περίπτωση (όταν δηλαδή ο εκθέτης είναι διαφορετικός από το μηδέν) η δύναμη υπολογίζεται μέσω του κώδικα :

```
for(j=0; j<power; j++)
        result = result * myBase;
```

Δηλαδή, η βάση πολλαπλασιάζεται με τον εαυτό της όσες φορές δηλώνει ο εκθέτης (μεταβλητή **power**).

Το παρακάτω κομμάτι κώδικα υλοποιεί την συνάρτηση f(x)=4/(1+x^2) της οποίας το ολοκλήρωμα τελικά υπολογίζεται.

```
double f(double x)
{
        double result = 0;
        result = 4/(1+myPow(x,2));
        return result;
}
```

Στην μεταβλητή **result** αποθηκεύεται το αποτέλεσμα που επιστρέφει η συνάρτηση για δεδομένο **x.** Η δύναμη x^2 υπολογίζεται μέσω της συνάρτησης myPow.

Όπως προαναφέρθηκε, σύμφωνα με τον κανόνα του Simpson έχουμε :

$$\int_a^b f(x)dx \cong \frac{h}{3}\left[f(x_0)+4f(x_1)+f(x_2)\right]$$

όπου x_0=a, x_1=a+h, x_2=b αι h=(b-a)/2

Ο υπολογισμός του ολοκληρώματος της συνάρτησης f(x), με όρια ολοκλήρωσης από a έως b γίνεται μέσω της συνάρτησης **calculateArea.**

```
double calculateArea(double a, double b)
{
        double result = 0;

        double h = (b-a)/2;
        double x0 = a;
        double x1 = a + h;
        double x2 = b;

        result = (h/3) * (f(x0) + 4*f(x1) + f(x2));

        return result;
}
```

Η συνάρτηση δέχεται ως ορίσματα τις μεταβλητές **a** και **b** που αποτελούν τα όρια ολοκλήρωσης και επιστρέφει την τιμή της μεταβλητής **result**, όπου έχει αποθηκευτεί το αποτέλεσμα του ολοκληρώματος.

Παρακάτω περιγράφεται η λειτουργία της συνάρτησης **main**, που είναι και η βασική συνάρτηση της εφαρμογής.

```
int main(int argc, char* argv[])
{
        int calculations_for_each_slave;
        int calculations_remainder;

          int rank, size, i;
         MPI_Status status;
```

Στη μεταβλητή ***calculations_for_each_slave*** αποθηκεύεται ο ελάχιστος αριθμός υπολογισμών που θα εκτελεστούν από κάθε slave διεργασία/node. Στη μεταβλητή ***calculations_remainder*** αποθηκεύεται ο αριθμός των διεργασιών/nodes που θα εκτελέσουν έναν επιπλέον υπολογισμό (***calculations_for_each_slave + 1***).

Στην ακέραια μεταβλητή **rank** θα καταχωρήσουμε την τάξη της τρέχουσας διεργασίας, στη μεταβλητή **size** θα καταχωρήσουμε το πλήθος των διεργασιών του τρέχοντος Communicator, ενώ η μεταβλητή **i** χρησιμοποιείται για την αποθήκευση του ID (τάξης) της τρέχουσας διεργασίας. Αναγκαία είναι η χρήση της μεταβλητής **status** που ανήκει στον τύπο δεδομένων **MPI_Status** και χρησιμοποιείται από τη συνάρτηση **MPI_Recv**.

```
        border_a = atof(argv[1]);
        border_b = atof(argv[2]);
        subIntervals_number = atoi(argv[3]);
```

Το πρόγραμμα δέχεται τρία ορίσματα. Η τιμή του πρώτου ορίσματος (**argv[1]**), αφού μετατραπεί σε τύπο δεδομένων double (ενώ αρχικά ήταν const char *) μέσω της εντολής **atof,** αποθηκεύεται στη μεταβλητή **border_a**. Όμοια, η τιμή του δεύτερου ορίσματος (**argv[2]**) αποθηκεύεται στη μεταβλητή **border_b**. Η τιμή του τρίτου ορίσματος (**argv[3]**), αφού πρώτα μετατραπεί σε ακέραια μεταβλητή (integer) μέσω της εντολής **atoi**, αποθηκεύεται στη μεταβλητή **subIntervals_number**.

```
        interval_length = (border_b - border_a)/subIntervals_number;
```

Στη συνέχεια υπολογίζεται το μήκος κάθε υπο-διαστήματος στα οποία θα χωριστεί το διάστημα [border_a, border_b]. Το μήκος κάθε υποδιαστήματος ισούται με την διαφορά (border_b-border_a) δια τον αριθμό των υποδιαστημάτων.

```
        //initialization
        final_result = 0;
        temp_x1 = 0;
        temp_x2 = border_a;
```

Οι παραπάνω γραμμές κώδικα αρχικοποιούν την τιμή των μεταβλητών **final_result**, **temp_x1** και **temp_x2**.

```
MPI_Init(&argc, &argv);
```

Πριν τη χρήση οποιασδήποτε εντολής MPI, θα πρέπει να λάβει χώρα η αρχικοποίηση του περιβάλλοντος MPI, η οποία γίνεται χρησιμοποιώντας τη συνάρτηση **MPI_Init**.

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

Η τάξη της τρέχουσας διεργασίας και το συνολικό πλήθος των διεργασιών του προεπιλεγμένου Communicator που περιγράφεται από τη σταθερά **MPI_COMM_WORLD** αποθηκεύονται στις μεταβλητές **rank** και **size** τύπου integer αντίστοιχα.καλώντας τις παραπάνω συναρτήσεις.

```
if(size != 1)
{
        calculations_for_each_slave = subIntervals_number/(size-1);
        calculations_remainder = subIntervals_number%(size-1);
```

Πρώτα απ'όλα, ελέγχουμε αν ο αριθμός των διεργασιών είναι διαφορετικός από το 1, γιατί σε περίπτωση που ισούται με το 1, το πρόγραμμα θα εκτελεστεί σαν ένα συμβατικό πρόγραμμα που δεν χρησιμοποιεί τις λειτουργίες που προσφέρει το MPI. Σε περίπτωση λοιπόν που έχουμε περισσότερες από μια διεργασίες, έχουμε **size-1** slave διεργασίες που θα εκτελέσουν τους υπολογισμούς των ολοκληρωμάτων και μια βασική (master) διεργασία που θα «αναθέτει» στις υπόλοιπες τους υπολογισμούς που θα εκτελέσουν και στο τέλος θα λαμβάνει το αποτέλεσμα κάθε υπολογισμού.

Ο ελάχιστος αριθμός των υπολογισμών (**calculations_for_each_slave**) που θα εκτελέσει κάθε διεργασία ισούται με το πηλίκο της διαίρεσης του πλήθους των υποδιαστημάτων (**subIntervals_number**) με το (**size-1**) που είναι το συνολικό πλήθος των διεργασιών μείον την πρώτη διεργασία, η οποία όπως αναφέρθηκε προηγουμένως αναλαμβάνει μόνο την διαχείριση των υπολογισμών.

Επίσης, κάποιες διεργασίες θα εκτελέσουν έναν επιπλέον υπολογισμό. Το πλήθος των διεργασιών αυτών ισούται με το υπόλοιπο της διαίρεσης του πλήθους των υποδιαστημάτων (**subIntervals_number**) με το (**size-1**) και αποθηκεύεται στη μεταβλητή **calculations_remainder**.

Το παρακάτω κομμάτι κώδικα καθορίζει την συμπεριφορά της βασικής διεργασίας (με ID ίσο με το 0), η οποία θα αναλάβει την διαχείριση των υπολοίπων διεργασιών.

```
i = 0;
if ( rank == 0 )
{
        for(subIntervals_counter=1;
        subIntervals_counter<=subIntervals_number;
        subIntervals_counter++)
        {
```

Με χρήση του μετρητή **subIntervals_counter** η πρώτη διεργασία εκτελεί το παρακάτω κομμάτι κώδικα για αριθμό επαναλήψεων που ισούται με το πλήθος των υπο-διαστημάτων (**subIntervals_number**).

```
i++;
if(i == size)
        i = 1;
```

Η μεταβλητή **i** αποτελεί έναν μετρητή για τις διεργασίες. Η τιμή της αυξάνεται κάθε φορά που διατρέχεται ο βρόγχος του **for** και όταν φτάσουμε στην τελευταία διεργασία (***i == size***), ξεκινάμε ξανά από την διεγρασία με ID ίσο με 1.

```
temp_x1 = temp_x2;
temp_x2 = temp_x1 + interval_length;
```

Οι παραπάνω δύο γραμμές κώδικα υπολογίζουν τα όρια για το κάθε υποδιάστημα και αποθηκεύουν τις τιμές αυτές στις μεταβλητές **temp_x1** και **temp_x2** αντίστοιχα.

```
printf("\n      --- Master says : I send data for
        interval No.%d ---\n",subIntervals_counter);

MPI_Send(&temp_x1,1,MPI_DOUBLE,
i,1,MPI_COMM_WORLD);
printf("Master : temp_x1 = %g sent to slave node
No:%d\n",temp_x1,i);

MPI_Send(&temp_x2,1,MPI_DOUBLE,
i,1,MPI_COMM_WORLD);
printf("Master : temp_x2 = %g sent to slave node
No:%d\n",temp_x2,i);

MPI_Send(&final_result,1,MPI_DOUBLE,
i,1,MPI_COMM_WORLD);
```

Μέσω της εντολής **MPI_Send(&temp_x1,1,MPI_DOUBLE,i,1, MPI_COMM_WORLD)** η βασική διεργασία (με ID ίσο με το 0) στέλνει την τιμή του ορίου **temp_x1** ($1^{ο}$ όρισμα) που είναι τύπου **MPI_DOUBLE** ($3^{ο}$ όρισμα) στην διεργασία με ID ίσο με **i** ($4^{ο}$ όρισμα). Το $2^{ο}$ όρισμα (**1**) δηλώνει το πλήθος των στοιχείων που πρόκειται να σταλούν, το $5^{ο}$ όρισμα (**1**) αποτελεί την ετικέτα του μηνύματος που επιτρέπει την αναγνώριση του μηνύματος ανάμεσα σε άλλα και τέλος το $6^{ο}$ όρισμα (**MPI_COMM_WORLD**) δηλώνει τον Communicator στον οποίο ανήκουν οι διεργασίες αποστολέας και παραλήπτης. Όμοια, η εργασία με ID ίσο με 0 στέλνει στην διεργασία με ID ίσο με i, το άλλο όριο του υπο-διαστήματος (**temp_x2**) καθώς επίσης και την τρέχουσα τιμή του τελικού αποτελέσματος (**final_result**). Οι εντολές **printf** χρησιμοποιούνται για να τυπώσουμε στην οθόνη μηνύματα περιγραφής της εντολής που εκτελείται κάθε χρονική στιγμή.

```
                MPI_Recv(&final_result,1,MPI_DOUBLE,
                i,2,MPI_COMM_WORLD,&status);
                printf("Master : I RECEIVED final_result = %g from
                slave node No.%d\n",final_result,i);
```

Αφού η βασική διεργασία έχει στείλει τα απαιτούμενα για τον υπολογισμό του ολοκληρώματος με όρια **temp_x1**, **temp_x2** στην διεργασία **i**, στη συνέχεια περιμένει να λάβει την ανανεωμένη τιμή του τελικού αποτελέσματος (**final_result**). Μέσω της εντολής **MPI_Recv (&final_result,1,MPI_DOUBLE,i,2,MPI_COMM_WORLD,&status)** η βασική διεργασία λαμβάνει την τιμή της μεταβλητής **final_result** (1ο όρισμα) που είναι **1** (2ο όρισμα) μεταβλητή τύπου **MPI_DOUBLE** (διπλής ακρίβειας) (3ο όρισμα), από την διεργασία με ID ίσο με **i** (4ο όρισμα). Το **2** (5ο όρισμα) αποτελεί την ετικέτα του μηνύματος, το 6ο όρισμα (**MPI_COMM_WORLD)** δηλώνει τον Communicator στον οποίο ανήκουν οι διεργασίες αποστελέας και παραλήπτης και τέλος η μεταβλητή **status** (7ο όρισμα) περιέχει πληροφορίες σχετικά με τη διαδικασία παραλαβής του μηνύματος,

```
        }
        printf("\n\n\n\n--- THE FINAL RESULT IS : %g ---
        \n\n\n\n",final_result);
}
```

Στη συνέχεια τυπώνεται στην οθόνη η τιμή του τελικού αποτελέσματος (**final_result**) για την τρέχουσα χρονική στιγμή.

Όλες οι υπόλοιπες διεργασίες (slave) πέραν της πρώτης (master), λαμβάνουν τα δεδομένα που τους στέλνει η πρώτη διεργασία, κάνουν τους απαραίτητους υπολογισμούς και στη συνέχεια επιστρέφουν στην πρώτη διεργασία το αποτέλεσμα.

```
else
{
        for(slave_calculations_counter=0;
        slave_calculations_counter<calculations_for_each_slave;
        slave_calculations_counter++)
        {
                MPI_Recv(&temp_x1,1,MPI_DOUBLE,
                0,1,MPI_COMM_WORLD,&status);
                printf("slave node No.%d : I received temp_x1 =
                %g\n",rank,temp_x1);

                MPI_Recv(&temp_x2,1,MPI_DOUBLE,
                0,1,MPI_COMM_WORLD,&status);
                printf("slave node No.%d : I received temp_x2 =
                %g\n",rank,temp_x2);

                MPI_Recv(&final_result,1,MPI_DOUBLE,
                0,1,MPI_COMM_WORLD,&status);
```

Κάθε "slave" διεργασία (με ID >= 1) εκτελεί το λιγότερο τόσους υπολογισμούς όσους ορίζει η μεταβλητή **calculations_for_each_slave**. Μέσω των παραπάνω τριών εντολών **MPI_Recv** η τρέχουσα slave διεργασία λαμβάνει από την "master" διεργασία (με ID ίσο με 0) τις τιμές των μεταβλητών **temp_x1**, **temp_x2** και **final_result**.

```
final_result = final_result + calculateArea(temp_x1,
temp_x2);
```

Οι μεταβλητές **temp_x1** και **temp_x2** ορίζουν τα όρια ολοκλήρωσης του ολοκληρώματος που καλείται να υπολογίσει η διεργασία. Μέσω της συνάρτησης **calculateArea** υπολογίζεται η τιμή του συγκεκριμένου ολοκληρώματος και προστίθεται στην τρέχουσα τιμή του τελικού αποτελέσματος (**final_result**).

```
printf("slave node No.%d : I SEND TO MASTER
final_result = %g\n",rank,final_result);

MPI_Send(&final_result,1,MPI_DOUBLE,
0,2,MPI_COMM_WORLD);
}
```

Στη συνέχεια, η νέα τιμή της μεταβλητής **final_result** αποστέλλεται στην πρώτη (master) διεργασία.

Κάποιες διεργασίες θα υπολογίσουν ένα επιπλέον ολοκλήρωμα, δηλαδή θα εκτελέσουν συνολικά **calculations_for_each_slave + 1** υπολογισμούς. Το πλήθος των διεργασιών αυτών ισούται με την τιμή της μεταβλητής **calculations_remainder**.

```
for(slave_calculations_counter=1;
slave_calculations_counter<=calculations_remainder;
slave_calculations_counter++)
{
```

Η μεταβλητή **slave_calculations_counter** παίρνει τιμές από 1 μέχρι την τιμή της μεταβλητής **calculations_remainder**.

```
if(rank == slave_calculations_counter)
{
    MPI_Recv(&temp_x1,1,MPI_DOUBLE,
    0,1,MPI_COMM_WORLD,&status);
    printf("slave node No.%d : I received
    temp_x1 = %g\n",rank,temp_x1);


    MPI_Recv(&temp_x2,1,MPI_DOUBLE,
    0,1,MPI_COMM_WORLD,&status);
    printf("slave node No.%d : I received
    temp_x2 = %g\n",rank,temp_x2);

    MPI_Recv(&final_result,1,MPI_DOUBLE,
    0,1,MPI_COMM_WORLD,&status);
```

Όταν το ID της διεργασίας ισούται με την τιμή του μετρητή **slave_calculations_counter**, η συγκεκριμένη διεργασία λαμβάνει, όπως προηγουμένως, τις τιμές των μεταβλητών **temp_x1** και **temp_x2** που αποτελούν τα όρια ολοκλήρωσης, καθώς επίσης και την τρέχουσα τιμή της μεταβλητής **final_result**.

```
final_result = final_result + calculateArea(temp_x1, temp_x2);

printf("slave node No.%d : I SEND TO MASTER final_result = %g\n",rank,final_result);

MPI_Send(&final_result,1,MPI_DOUBLE,0,2,MPI_COMM_WORLD);
```

Στη συνέχεια υπολογίζεται η τιμή του ολοκληρώματος (***calculateArea(temp_x1, temp_x2)***), το αποτέλεσμα προστίθεται στην τιμή της μεταβλητής **final_result** και κατόπιν η νέα τιμή της **final_result** αποστέλλεται στην βασική διεργασία.

```
                }
            }
        }
    }
```

Όταν το πλήθος των διεργασιών ισούται με 1, αυτό σημαίνει ότι θα πρέπει όλοι οι υπολογισμοί να εκτελεστούν από τη μοναδική διαθέσιμη διεργασία. Στην περίπτωση αυτή, δεν χρησιμοποιείται καμία λειτουργία του MPI.

```
    else
    {
        for(subIntervals_counter=1;
        subIntervals_counter<=subIntervals_number;
        subIntervals_counter++)
        {
            temp_x1 = temp_x2;
            temp_x2 = temp_x1 + interval_length;
            final_result = final_result + calculateArea(temp_x1, temp_x2);
        }
        printf("\n\n\n\n--- THE FINAL RESULT IS : %g --- \n\n\n\n",final_result);
    }
```

Για κάθε υπο-διάστημα υπολογίζονται τα όρια **temp_x1** και **temp_x2**. Η τιμή της μεταβλητής **temp_x1** γίνεται ίση με την τρέχουσα τιμή του **temp_x2** και στη συνέχεια η τιμή της μεταβλητής **temp_x2** γίνεται ίση με το άθροισμα της μεταβλητής **temp_x1** συν το μήκος κάθε υποδιαστήματος (**interval_length**). Υπολογίζεται το ολοκλήρωμα της f(x) με όρια ολοκλήρωσης τα temp_x2 και temp_x2 και η τιμή του προστίθεται στο τελικό αποτέλεσμα. Αφού ολοκληρωθεί αυτή η επαναληπτική διαδικασία, τυπώνεται στην οθόνη το τελικό αποτέλεσμα.

```
    MPI_Finalize();
```

Με την κλήση της **MPI_Finalize** απελευθερώνεται η μνήμη που δεσμεύεται από τις δομές δεδομένων του προτύπου MPI και γενικότερα τερματίζεται η λειτουργία του.

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

Η εντολή return(0) τερματίζει την λειτουργία του προγράμματος.

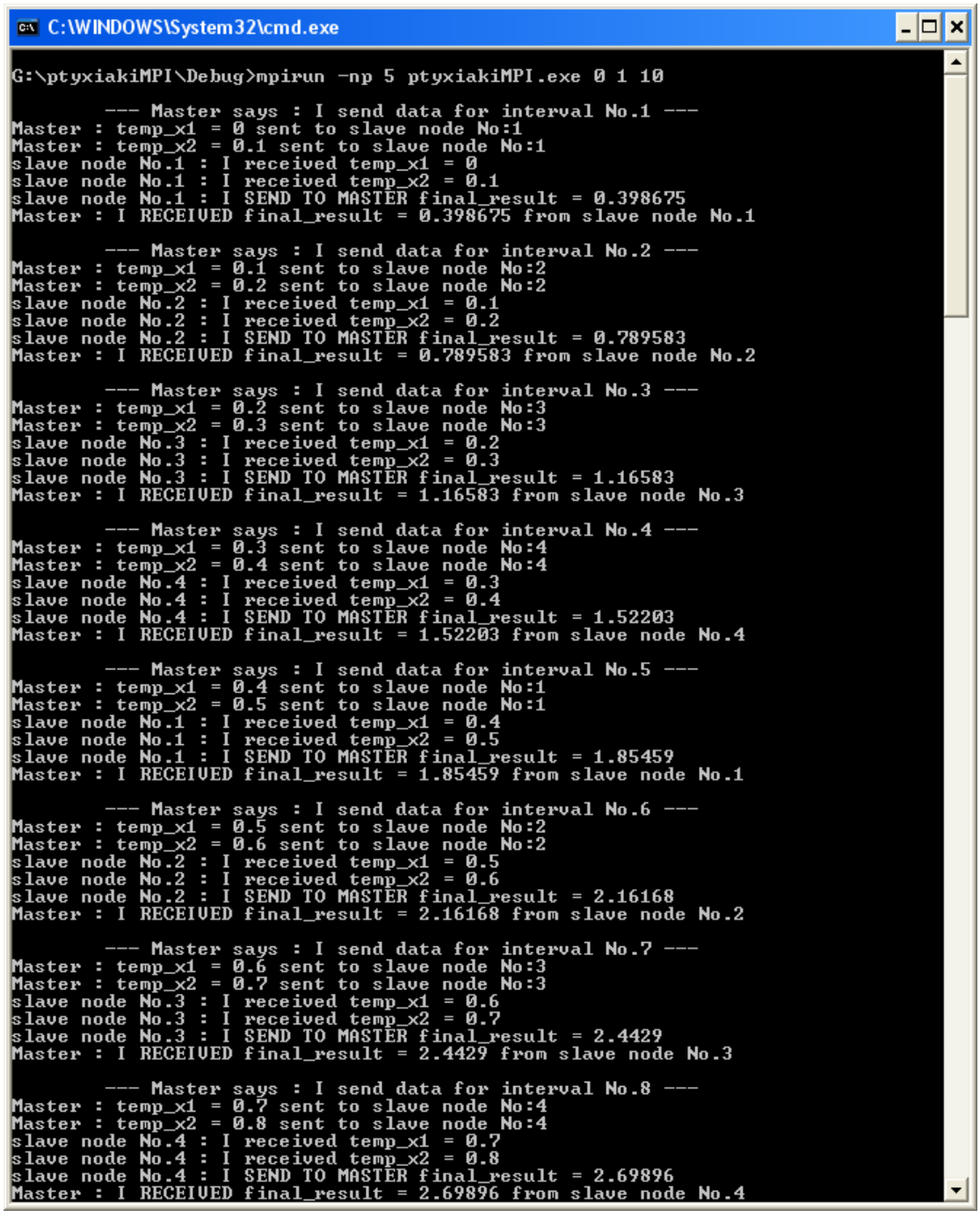
4.1.2. Παράδειγμα εκτέλεσης του προγράμματος

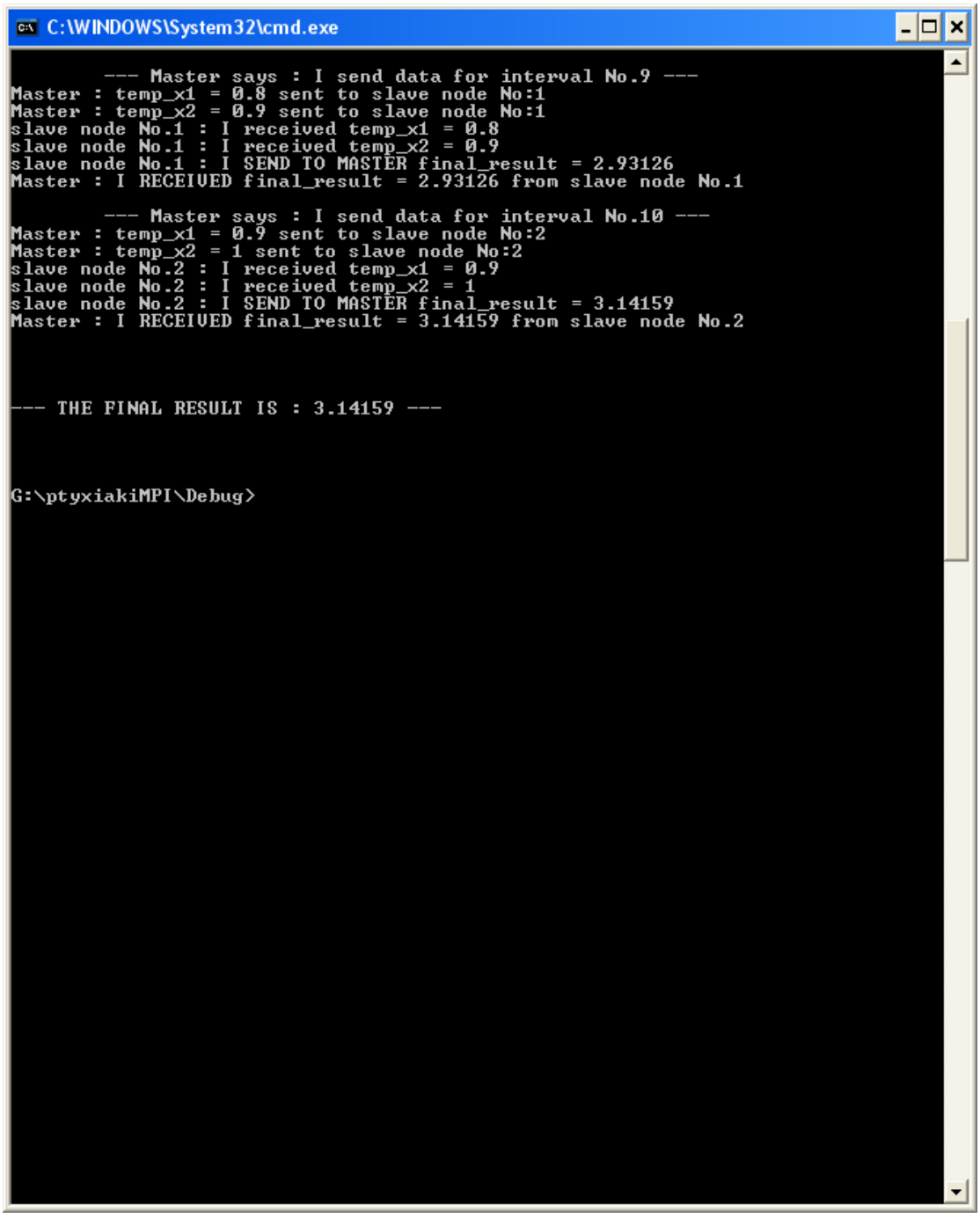
Αν εκτελέσουμε το πρόγραμμα με τα εξής ορίσματα :

mpirun -np 5 ptyxiakiMPI.exe 0 1 10

εννοούμε ότι ο υπολογισμός του ολοκληρώματος της συνάρτησης f(x) θα γίνει από 5 διεργασίες/nodes, τα όρια ολοκλήρωσης θα είναι από 0 εως 1 και ότι το διάστημα [a, b] ([0, 1]) θα χωριστεί σε 10 υπο-διαστήματα.

Παρακάτω παρουσιάζεται το αποτέλεσμα της εκτέλεσης του προγράμματος.




C:\WINDOWS\System32\cmd.exe
--- Master says : I send data for interval No.9 ---
Master : temp_x1 = 0.8 sent to slave node No:1
Master : temp_x2 = 0.9 sent to slave node No:1
slave node No.1 : I received temp_x1 = 0.8
slave node No.1 : I received temp_x2 = 0.9
slave node No.1 : I SEND TO MASTER final_result = 2.93126
Master : I RECEIVED final_result = 2.93126 from slave node No.1
--- Master says : I send data for interval No.10 ---
Master : temp_x1 = 0.9 sent to slave node No:2
Master : temp_x2 = 1 sent to slave node No:2
slave node No.2 : I received temp_x1 = 0.9
slave node No.2 : I received temp_x2 = 1
slave node No.2 : I SEND TO MASTER final_result = 3.14159
Master : I RECEIVED final_result = 3.14159 from slave node No.2
--- THE FINAL RESULT IS : 3.14159 ---
G:\ptyxiakiMPI\Debug>

ΚΕΦΑΛΑΙΟ 5

5. Η μέθοδος Monte Carlo

Η μέθοδος Monte Carlo αποτελεί μια από τις πιο χαρακτηριστικές εφαρμογές φυσικού παραλληλισμού. Σύμφωνα με τη μέθοδο αυτή, χρησιμοποιούνται τυχαίοι αριθμοί και στατιστική ανάλυση για τον υπολογισμό αριθμητικών και φυσικών προβλημάτων με κάθε ένα από τα βήματα του υπολογισμού να είναι ανεξάρτητο από τα υπόλοιπα. Το όνομα Monte Carlo προέρχεται από την ομοιότητα της στατιστικής ανάλυσης με τα τυχερά παιχνίδια (το Monte Carlo είναι ένα από τα μεγαλύτερα κέντρα τυχερών παιχνιδιών).

Η χρήση της μεθόδου Monte Carlo επιτρέπει την εξέταση πολύ πολύπλοκων μοντέλων. Για παράδειγμα, η επίλυση εξισώσεων οι οποίες περιγράφουν τις σχέσεις μεταξύ δύο ατόμων είναι σχετικά απλή. Η επίλυση όμως των ίδιων εξισώσεων για τις σχέσεις μεταξύ χιλιάδων ατόμων είναι σχεδόν αδύνατη. Με τη μέθοδο Monte Carlo ένα μεγάλο σύστημα μπορεί να χωριστεί σε έναν αριθμό τυχαίων επιπέδων, τα οποία μπορούν να περιγράψουν το σύστημα στο σύνολό του.

Η μέθοδος Monte Carlo βασίζεται στο γεγονός ότι ενώ το αποτέλεσμα ενός μοναδικού τυχαίου γεγονότος δεν είναι προβλέψιμο, το αποτέλεσμα μιας σειράς τυχαίων γεγονότων μπορεί να προβλεφθεί. Για παράδειγμα, κατά την ρίψη ενός νομίσματος, ενώ είναι αδύνατη η πρόβλεψη του αποτελέσματος μιας τυχαίας ρίψης, είναι πολύ πιθανόν ότι ύστερα από έναν μεγάλο αριθμό ρίψεων ο αριθμός των κορώνων θα είναι ίδιος με τον αριθμό των γραμμάτων. Από τη στιγμή που υπάρχουν δύο πιθανά αποτελέσματα, το καθένα με ίσες πιθανότητες, λέμε ότι η αναμενόμενη διανομή των αποτελεσμάτων είναι 50/50.

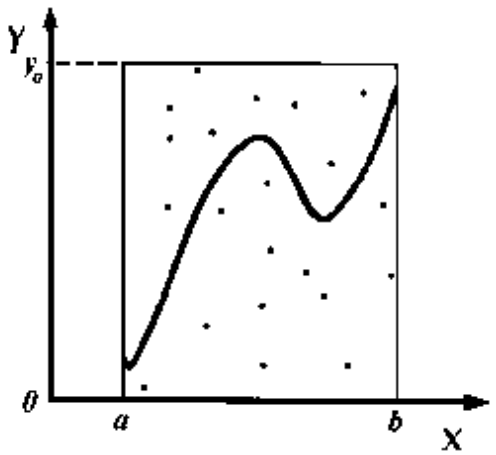
5.1. Εφαρμογές της μεθόδου Monte Carlo

Υπολογισμός ολοκληρωμάτων

Ο υπολογισμός του ολοκληρώματος μιας συνάρτησης f(x) αποτελεί την απλούστερη δυνατή εφαρμογή της μεθόδου Monte Carlo.

$$I = \int_a^b f(x)dx \tag{5.1.1}$$

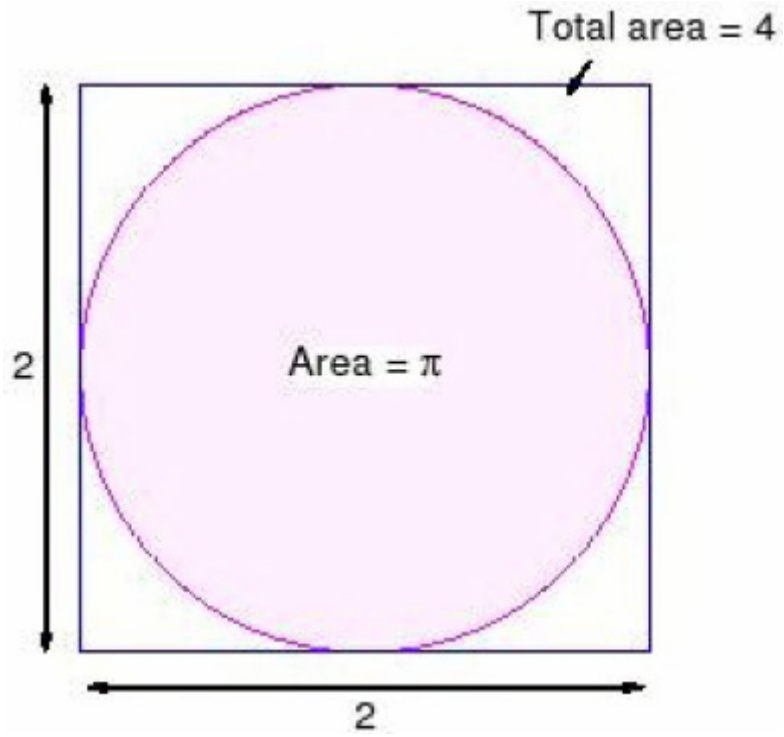
Για να υπολογίσουμε το εμβαδό που καλύπτει μια συνάρτηση, σχεδιάζουμε μια περιοχή που περικλείει το ζητούμενο εμβαδό. Από την περιοχή αυτή επιλέγουμε N τυχαία σημεία που προέρχονται από ομοιόμορφη κατανομή. Μεταξύ των σημείων αυτών υπάρχουν N_0 που πέφτουν στην περιοχή του οποίου το εμβαδόν θέλουμε να υπολογίσουμε. Ο λόγος του πλήθους των σημείων που έπεσαν κάτω από την καμπύλη της f(x) προς το συνολικό πλήθος των σημείων, N_0/N, δίνει το ποσοστό από το εμβαδό του παραλληλογράμμου που αντιστοιχεί στο ολοκλήρωμα (5.1.1).



Σχήμα 24. *Η ρίψη N σημείων εντός του παραλληλογράμμου δίνει N_0 σημεία στην περιοχή κάτω από την συνάρτηση f(x). Το ολοκλήρωμα της f(x) στο διάστημα [a, b] δίνεται τότε από την (5.1.2).*

$$I = \frac{N_0}{N}[y_0(b-a)] \tag{5.1.2}$$

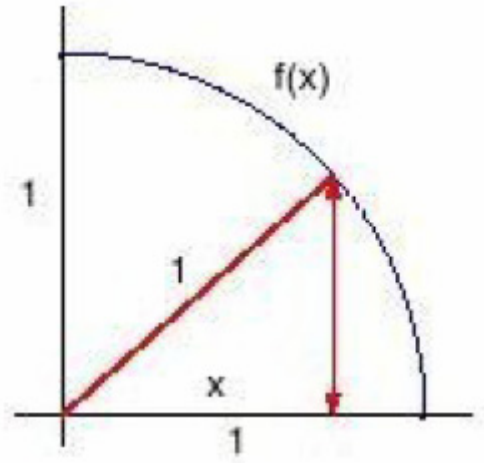
Υπολογισμός της τιμής του π



Σχήμα 25. *Υπολογισμός του π με τη μέθοδο Monte Carlo*

Μια από τις πιο γνωστές εφαρμογές της μεθόδου Monte Carlo είναι ο υπολογισμός του αριθμού π. Ένα τετράγωνο εφάπτεται σε έναν κύκλο με την κάθε πλευρά του τετραγώνου να είναι 2 φορές μεγαλύτερη από την ακτίνα του κύκλου, όπως φαίνεται στο σχήμα 25.

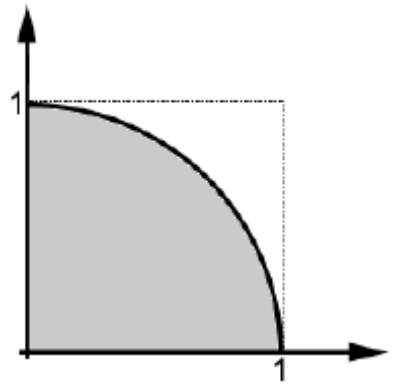
Για να εξετάσουμε το πρόβλημα, ας επιλέξουμε το πρώτο τεταρτημόριο του κύκλου αυτού:



Σχήμα 26. *Το πρώτο τεταρτημόριο του κύκλου*

Για να λειτουργήσει η μέθοδος Monte Carlo, επιλέγονται τυχαία σημεία εντός του τετραγώνου και σημειώνεται ο αριθμός αυτών που βρίσκονται εντός του κύκλου. Το κλάσμα των σημείων τα οποία βρίσκονται εντός του κύκλου προς τα συνολικά σημεία θα είναι π/4. Για να υπολογίσουμε το π, αρκεί να υπολογίσουμε το εμβαδόν του τεταρτοκυκλίου :

$$E = \frac{1}{4}\pi r^2 \Rightarrow E = \frac{1}{4}\pi \Rightarrow \pi = 4E \quad (5.1.3)$$



Σχήμα 27. *Το σκιασμένο εμβαδόν είναι ανάλογο του πλήθους των τυχαίων σημείων που πέφτουν μέσα στην περιοχή αυτή προς το σύνολο των σημείων που πέφτουν στο τετράγωνο που ορίζουν οι άξονες και οι διακεκομμένες γραμμές.*

Η ρίψη τυχαίων σημείων στην περιοχή του τετραγώνου μας δίνει το σκιασμένο εμβαδόν, που ισούται με :

$$E = \frac{N_0}{N} E_{\text{τετρ}} \Rightarrow E = \frac{N_0}{N} \quad (5.1.4)$$

Εάν υποθέσουμε ότι η ακτίνα του κύκλου ισούται με 1, για κάθε σημείο επιλέγονται δύο τυχαίοι αριθμοί, ένας ως συντεταγμένη x και ένας ως συντεταγμένη y, τα οποία μπορούν να χρησιμοποιηθούν για τον υπολογισμό της απόστασης του σημείου από το κέντρο του κύκλου (0, 0) χρησιμοποιώντας το Πυθαγόρειο θεώρημα. Αν η απόσταση αυτή είναι μικρότερη ή ίση του 1, τότε το σημείο θα βρίσκεται εντός του κύκλου. Όταν εκτελεστεί η διαδικασία αυτή πολλές φορές, μπορεί να υπολογιστεί η τιμή του π με την ακρίβεια του αποτελέσματος να είναι ευθέως ανάλογη του πλήθους των επαναλήψεων.

Η ακρίβεια ενός Monte Carlo υπολογισμού έχει μεγάλη εξάρτηση από το πλήθος Ν των σημείων. Μεγάλη εξάρτηση έχει επίσης και από την ποιότητα της γεννήτριας των τυχαίων αριθμών.

5.2. Υλοποίηση της Μεθόδου Monte Carlo με χρήση του MPI - Υπολογισμός της τιμής του π

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <stdlib.h>


int main( int argc, char *argv[])
{
    int intervals_num, rank, size, i;
    double PI25DT = 3.14159265358979323846264643;
    double startwtime = 0.0, endwtime;
    double *random_x, *random_y;
    int pieces;
    float in_square=0.0, in_circle=0.0;
    float total_squares=0.0, total_circles=0.0;
    float pi;

    /* Initialize (start) MPI */
    MPI_Init(&argc,&argv);

    /* How many nodes/processes are available? who am I? */
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    /* I am the master node/process */
    if(rank == 0)
    {
        printf("Enter the number of intervals:");
        scanf("%d",&intervals_num);

        /* Start the timer */
        startwtime = MPI_Wtime();

        /* Determine how many pieces each proccess will calculate */
        pieces=intervals_num/size;
    }

    /* Broadcast the number of pieces to let proccesses allocate space */
    MPI_Bcast(&pieces, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* Allocate space */
    random_x = malloc(pieces * sizeof(double));
    random_y = malloc(pieces * sizeof(double));

    /* Generate random numbers */
    srand(time(NULL) + rank);
    for(i=0; i<pieces; i++)
    {
        random_x[i] = (double)rand()/(double)RAND_MAX; /* use 1/4 circle */
        random_y[i] = (double)rand()/(double)RAND_MAX;
    }

    /* Calculate hits */
    for(i=0; i<pieces; i++)
    {
        in_square++;
        if(random_x[i]*random_x[i] + random_y[i]*random_y[i] <= 1.0)
            in_circle++;
    }

    /* sum all the values to get total number of hits */
    MPI_Reduce(&in_square, &total_squares, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&in_circle, &total_circles, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

    if(rank == 0)
    {
        printf("Total squares: %.0f Total circles: %.0f\n",total_squares, total_circles);
        pi = 4.0*total_circles / total_squares;
        endwtime = MPI_Wtime();
        printf("pi is approximately %.16f\n", pi);
        printf("error is %.16f\n", fabs(pi - PI25DT));
        printf("elapsed time is %f\n", endwtime-startwtime);
    }

    MPI_Finalize();
    return 0;
}
```

5.2.1. Επεξήγηση του πηγαίου κώδικα

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <stdlib.h>


int main( int argc, char *argv[])
{
        int intervals_num, rank, size, i;
```

Στην ακέραια μεταβλητή **intervals_num** θα καταχωρήσουμε τον αριθμό των υποδιαστηνάτων στα οποία θα χωριστεί το τεταρτημόριο του κύκλου, στη μεταβλητή **rank** θα καταχωρήσουμε την τάξη της τρέχουσας διεργασίας, στη μεταβλητή **size** το πλήθος των διεργασιών του τρέχοντος Communicator, ενώ η μεταβλητή **i** χρησιμοποιείται ως μετρητής.

```
        double PI25DT = 3.14159265358979323846264 3;
```

Στη μεταβλητή **PI25DT** αποθηκεύουμε τα πρώτα 25 ψηφία του αριθμού π, ώστε κατά την λήξη εκτέλεσης του προγράμματος να υπολογίσουμε το σφάλμα του υπολογισμού.

```
        double startwtime = 0.0, endwtime;
```

Στις μεταβλητές **startwtime** και **endwtime** θα αποθηκευτούν οι χρόνοι έναρξης και λήξης του προγράμματος αντίστοιχα.

```
        double *random_x, *random_y;
```

Οι μεταβλητές **random_x** και **random_y** αποτελούν δύο δείκτες σε αριθμούς κινητής υποδιαστολής.

```
        int pieces;
```

Στην ακέραια μεταβλητή **pieces** καταχωρείται το πλήθος των υπολογισμών που θα εκτελέσει κάθε διεργασία.

```
        float in_square=0.0, in_circle=0.0;
```

Η μεταβλητή **in_square** αποτελεί για κάθε διεργασία έναν μετρητή των σημείων που βρίσκονται μέσα στο τετράγωνο (σχήμα 27), ενώ η μεταβλητή **in_circle** χρησιμοποιείται ως μετρητής των σημείων που βρίσκονται μέσα στον κύκλο.

```
        float total_squares=0.0, total_circles=0.0;
        float pi;
```

Στη μεταβλητή **pi** θα αποθηκευτεί το τελικό αποτέλεσμα (η τιμή του π) ενώ στις μεταβλητές **total_squares** και **total_circles** πρόκειται να αποθηκευτεί το πλήθος των σημείων που βρίσκονται εντός του τετραγώνου και εντός του κύκλου (σχήμα 27) αντίστοιχα. Στη μεταβλητή **pi** θα

καταχωρηθεί η τιμή του π, όπως αυτή θα υπολογιστεί με χρήση της μεθόδου Monte Carlo.

```
/* Initialize (start) MPI */
MPI_Init(&argc,&argv);
```

Πριν τη χρήση οποιασδήποτε εντολής MPI, λαμβάνει χώρα η αρχικοποίηση του περιβάλλοντος MPI, η οποία γίνεται χρησιμοποιώντας τη συνάρτηση **MPI_Init**.

```
/* How many nodes/processes are available? who am I? */
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Η τάξη της τρέχουσας διεργασίας και το συνολικό πλήθος των διεργασιών του προεπιλεγμένου Communicator που περιγράφεται από τη σταθερά **MPI_COMM_WORLD** αποθηκεύονται στις μεταβλητές **rank** και **size** τύπου integer αντίστοιχα.καλώντας τις παραπάνω συναρτήσεις.

Το παρακάτω κομμάτι κώδικα καθορίζει την συμπεριφορά της βασικής διεργασίας (με ID ίσο με το 0).

```
/* I am the master node/process */
if(rank == 0)
{
        printf("Enter the number of intervals:");
        scanf("%d",& intervals_num);
```

Με την εντολή **printf** τυπώνεται στην οθόνη η φράση *"Enter the number of intervals:"* και στη συνέχεια μέσω της εντολής **scanf** η τιμή που θα εισάγει ο χρήστης αποθηκεύεται στην ακέραια μεταβλητή **intervals_num**.

```
        /* Start the timer */
        startwtime = MPI_Wtime();
```

Με την κλήση της εντολής **MPI_Wtime** επιστρέφεται η τιμή της τρέχουσας χρονικής στιγμής και η τιμή αυτή αποθηκεύεται στη μεταβλητή **startwtime**. Δεδομένου ότι βρισκόμαστε μέσα στο μπλοκ κώδικα που ορίζει η συνθήκη **if(rank == 0)**, στη μεταβλητή **startwtime** αποθηκεύεται η τιμή της χρονικής στιγμής έναρξης της διεργασίας με ID ίσο με το 0.

```
        /* Determine how many pieces each proccess will calculate */
        pieces= intervals_num/size;
```

Στη συνέχεια, υπολογίζεται ο αριθμός των υποδιαστημάτων που θα υπολογίσει κάθε διεργασία και αποθηκεύεται στην ακέραια μεταβλητή **pieces**. Ο αριθμός των υποδιαστημάτων που θα υπολογιστούν από κάθε διεργασία ισούται με το πηλίκο της διαίρεσης του αριθμού των υποδιαστημάτων δια του πλήθους των διεργασιών.

```
}
```

```
/* Broadcast the number of pieces to let proccesses allocate space */
MPI_Bcast(&pieces, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Η εντολή **MPI_Bcast** αποστέλλει ένα μήνυμα από κάποια διεργασία – που ονομάζεται διεργασία ρίζα (root process) – σε όλες τις υπόλοιπες διεργασίες της τρέχουσας ομάδας. Η χρήση της συνάρτησης **MPI_Bcast** χαρακτηρίζεται από μία σύνταξη της μορφής int MPI_Bcast (void * buffer, int count, MPI_Datatype dataType, int root, MPI_Comm comm). Στην παραπάνω σύνταξη, τα τρία πρώτα ορίσματα ταυτοποιούν το μήνυμα που αποστέλλεται: το πρώτο όρισμα αναφέρεται στην περιοχή μνήμης που περιέχει το εν λόγω μήνυμα, ενώ το δεύτερο και το τρίτο όρισμα επιτρέπουν τον καθορισμό του πλήθους και του τύπου δεδομένων των στοιχείων που περιλαμβάνονται στο μήνυμα. Στη συνέχεια η συνάρτηση περιέχει την τάξη της διεργασίας που θα αναλάβει την αποστολή του μηνύματος (το όρισμα root) ενώ το τελευταίο όρισμα αναφέρεται κατά τα γνωστά στον communicator που χρησιμοποιείται για τη διακίνηση των μηνυμάτων της εφαρμογής. Είναι ενδιαφέρον να αναφερθεί πως η συνάρτηση **MPI_Bcast** λειτουργεί με τέτοιο τρόπο, ώστε κατά τον τερματισμό της λειτουργίας της να έχει λάβει χώρα αντιγραφή των δεδομένων του μηνύματος στις περιοχές αποθήκευσης των υπόλοιπων διεργασιών του συστήματος, έτσι ώστε να μην είναι αναγκαία η κλήση κάποια συνάρτησης παραλαβής, όπως είναι για παράδειγμα η **MPI_Recv**.

Στην περίπτωσή μας, μέσω της εντολής **MPI_Bcast**, η βασική διεργασία (με ID ίσο με 0) αποστέλλει σε όλες τις υπόλοιπες διεργασίες την τιμή της ακέραιας μεταβλητής **pieces**, η οποία, όπως προαναφέρθηκε, δηλώνει τον αριθμό των υποδιαστημάτων που καλείται να υπολογίσει κάθε διεργασία.

```
/* Allocate space */
random_x = malloc(pieces * sizeof(double));
random_y = malloc(pieces * sizeof(double));
```

Για καθεμιά από τις μεταβλητές **random_x** και **random_y** δεσμεύεται με χρήση της εντολής **malloc** χώρος για τόσους αριθμούς κινητής υποδιαστολής όσοι είναι και οι υπολογισμοί που θα εκτελεστούν από κάθε διεργασία.

```
/* Generate random numbers */
srand(time(NULL) + rank);
```

Η εντολή **srand** χρησιμοποιείται για την αρχικοποίηση ενός μηχανισμού παραγωγής τυχαίων αριθμών. Η εντολή **time(NULL)** επιστρέφει την ώρα του ρολογιού του υπολογιστή. Το όρισμα **(time(NULL) + rank)** είναι μοναδικό για κάθε κλήση της **srand** και το γεγονός αυτό διασφαλίζει την παραγωγή μοναδικών αριθμών σε κάθε κλήση της εντολής **rand** που θα ακολουθήσει.
Σημείωση *: Η παραγωγή μοναδικών αριθμών αποτελεί πολύ σημαντικό γεγονός, καθώς επηρεάζει την ακρίβεια της μεθόδου Monte Carlo.*

```
for(i=0; i<pieces; i++)
{
        random_x[i] = (double)rand()/(double)RAND_MAX; /* use 1/4
circle */
```

```
        random_y[i] = (double)rand()/(double)RAND_MAX;
}
```

Στους δείκτες **random_x** και **random_y**, οι οποίοι αποτελούν μονοδιάστατους πίνακες αριθμών κινητής υποδιαστολής, αποθηκεύονται τόσοι τυχαίοι αριθμοί κινητής υποδιαστολής όσοι είναι και οι υπολογισμοί που θα εκτελεστούν από κάθε διεργασία. Κάθε τυχαίος αριθμός ισούται με το πηλίκο του αριθμού που επιστρέφει η συνάρτηση **rand** δια τον μέγιστο αριθμό (**RAND_MAX)** που είναι δυνατό να επιστραφεί από την συνάρτηση **rand**.

```
/* Calculate hits */
for(i=0; i<pieces; i++)
{
        in_square++;
        if(random_x[i]*random_x[i] + random_y[i]*random_y[i] <= 1.0)
                in_circle++;
}
```

Για κάθε υπολογισμό που εκτελεί κάθε διεργασία αυξάνεται κατά ένα ο μετρητής **in_square**, επειδή σε κάθε περίπτωση το σημείο που επιλέγεται βρίσκεται εντός του τετραγώνου (Σχήμα 28). Η συνθήκη **if(random_x[i]*random_x[i] + random_y[i]*random_y[i] <= 1.0)**, βάσει του Πυθαγορείου θεωρήματος καθορίζει αν το τυχαίο σημείο (random_x[i], random_y[i]) βρίσκεται εντός του κύκλου. Σε περίπτωση που το σημείο βρίσκεται μέσα στην περιοχή που ορίζει ο κύκλος, αυξάνεται κατά μια μονάδα η τιμή της μεταβλητής **in_circle**.

```
/* sum all the values to get total number of hits */
MPI_Reduce(&in_square, &total_squares, 1, MPI_FLOAT, MPI_SUM, 0,
MPI_COMM_WORLD);
```

Η συνάρτηση **MPI_Reduce** δέχεται ως είσοδο ένα πίνακα τιμών που έχουν αποσταλεί από άλλες διεργασίες, και υπολογίζει την τιμή μιας απλής ποσότητας, εφαρμόζοντας πάνω στις τιμές του διανύσματος εισόδου, μια αριθμητική ή λογική πράξη, που είτε προσφέρεται από το σύστημα, είτε δημιουργείται και χρησιμοποιείται από το χρήστη. Η χρήση της συνάρτησης MPI_Reduce χαρακτηρίζεται από μια σύνταξη της μορφής int MPI_Reduce (void * sendBuffer, void * recvBuffer, int count, MPI_Datatype dataType, MPI_Op operation, int root, MPI_Comm comm). Στην παραπάνω σύνταξη, το όρισμα sendBuffer αναφέρεται στην περιοχή μνήμης που περιέχει τα δεδομένα εισόδου της συνάρτησης, το πλήθος και ο τύπος των οποίων καθορίζονται από τα ορίσματα count και dataType. Το είδος της πράξης που εφαρμόζεται πάνω σε αυτά τα δεδομένα καθορίζεται από το όρισμα operation που ανήκει στον τύπο δεδομένων MPI_Op, ενώ το αποτέλεσμα που θα προκύψει αποθηκεύεται στην περιοχή μνήμης που περιγράφεται από το όρισμα recvBuffer. Τέλος το όρισμα root περιγράφει την τάξη (ID) της διεργασίας που θα πραγματοποιήσει την πράξη της αναγωγής, ενώ το όρισμα comm αναφέρεται κατά τα γνωστά στον communicator δια της χρήσεως του οποίου θα λάβει χώρα η διακίνηση των δεδομένων της εφαρμογής.

Επομένως, στην περίπτωσή μας, η παραπάνω κλήση της εντολής **MPI_Reduce** δηλώνει ότι η βασική διεργασία (με ID ίσο με 0) θα υπολογίσει το άθροισμα των τιμών της μεταβλητής **in_square** όπως αυτές επιστρέφονται από κάθε διεργασία και το άθροισμα αυτό θα αποθηκευτεί στην μεταβλητή **total_squares**. Άρα, τελικά, στην μεταβλητή **total_squares** θα έχει καταχωρηθεί ο συνολικός αριθμός των σημείων που βρίσκονται εντός του τετραγώνου (σχήμα 27).

```
MPI_Reduce(&in_circle, &total_circles, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Ομοίως, μέσω της παραπάνω συνάρτησης υπολογίζεται ο συνολικός αριθμός των σημείων που βρίσκονται εντός του κύκλου (σχήμα 27) και αποθηκεύεται στη μεταβλητή **total_circles**.

Στη συνέχεια, η βασική διεργασία (με ID/rank ίσο με 0) θα αναλαμβάνει να υπολογίσει την τιμή του π και να τυπώσει στην οθόνη διάφορα πληροφοριακά μηνύματα.

```
if(rank == 0)
{
        printf("Total squares: %.0f Total circles: %.0f\n",total_squares, total_circles);
```

Η παραπάνω εντολή **printf**, τυπώνει στην οθόνη τον συνολικό αριθμό των σημείων που βρέθηκαν εντός του τετραγώνου καθώς επίσης και τον συνολικό αριθμό των σημείων που βρέθηκαν εντός του κύκλου (σχήμα 27).

```
        pi = 4.0*total_circles / total_squares;
```

Μέσω της παραπάνω γραμμής κώδικα υπολογίζεται η τιμή του π βάσει των εξισώσεων (5.1.3) και (5.1.4) και η τιμή αυτή αποθηκεύεται στη μεταβλητή **pi**.

```
        endwtime = MPI_Wtime();
```

Με την κλήση της εντολής **MPI_Wtime** επιστρέφεται η τιμή της τρέχουσας χρονικής στιγμής και η τιμή αυτή αποθηκεύεται στη μεταβλητή **endwtime**. Δεδομένου ότι βρισκόμαστε μέσα στο μπλοκ κώδικα που ορίζει η συνθήκη **if(rank == 0)**, στη μεταβλητή **endwtime** θα αποθηκευτεί η τιμή της χρονικής στιγμής τερματισμού της βασικής διεργασίας (με ID ίσο με το 0).

```
        printf("pi is approximately %.16f\n", pi);
```

Η κλήση της παραπάνω εντολής **printf** έχει ως αποτέλεσμα το να τυπωθεί στην οθόνη η τιμή του π (το **16f** δηλώνει ότι θα τυπωθούν τα πρώτα 16 δεκαδικά ψηφία), όπως αυτή υπολογίστηκε μέσω της μεθόδου Monte Carlo.

```
        printf("error is %.16f\n", fabs(pi - PI25DT));
```

Η εντολή **fabs(pi - PI25DT)** υπολογίζει την απόλυτη τιμή της διαφοράς **(pi - PI25DT)**, που αποτελεί και το σφάλμα της μεθόδου Monte Carlo.

```
printf("elapsed time is %f\n", endwtime-startwtime);
```

Η παραπάνω εντολή **printf** τυπώνει στην οθόνη τον χρόνο που διήρκεσε η εκτέλεση του προγράμματος (**endwtime-startwtime**).

```
    }

    MPI_Finalize();
```

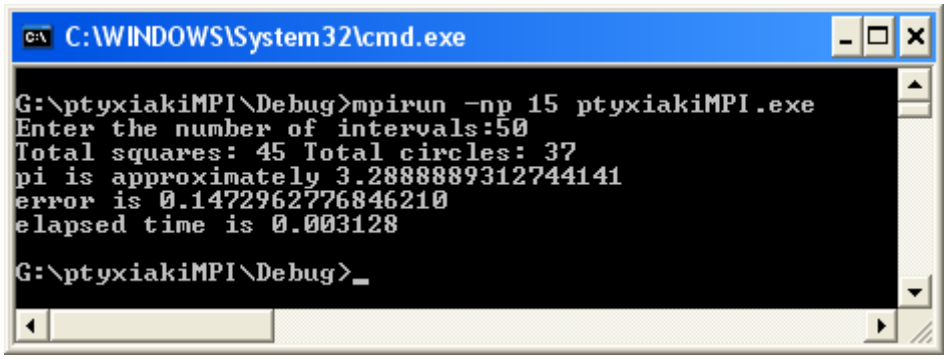
Με την κλήση της **MPI_Finalize** απελευθερώνεται η μνήμη που δεσμεύεται από τις δομές δεδομένων του προτύπου MPI και γενικότερα τερματίζεται η λειτουργία του.

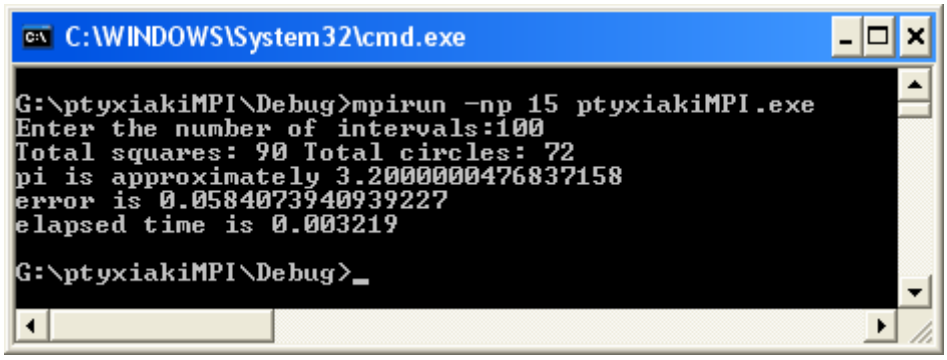
```
    return 0;
}
```

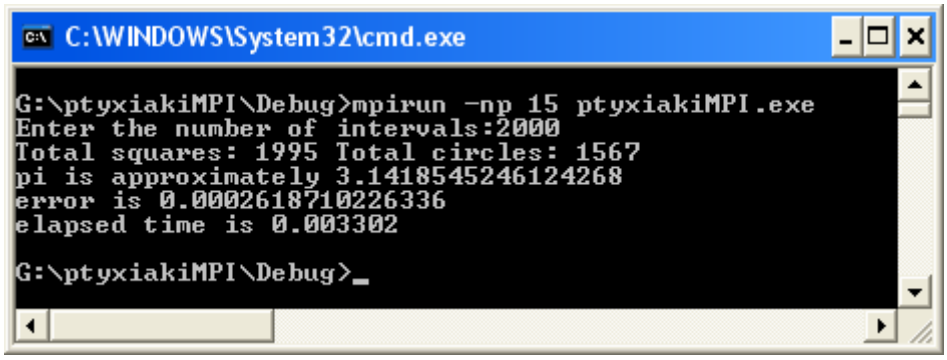
Η εντολή return(0) τερματίζει την λειτουργία του προγράμματος.

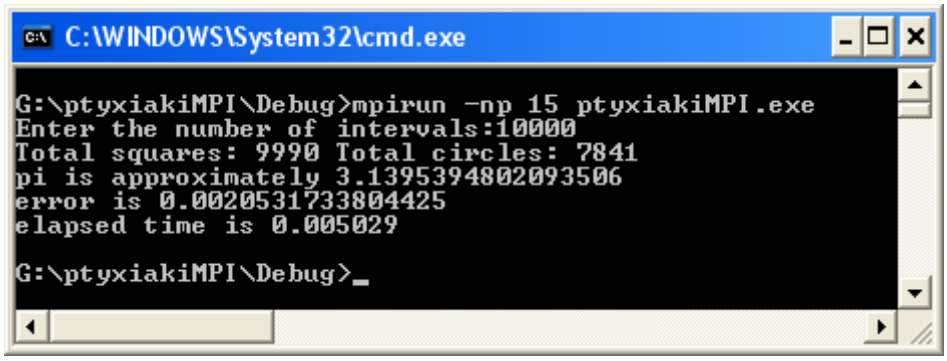
5.2.2. Παραδείγματα εκτέλεσης του προγράμματος

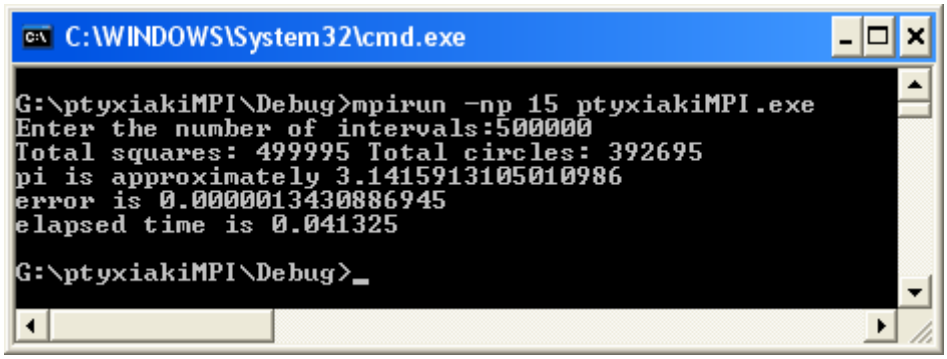
Παρακάτω παρουσιάζονται τα αποτελέσματα εκτέλεσης του προγράμματος για κάποιες ενδεικτικές τιμές (με αριθμό υποδιαστημάτων στα οποία χωρίζεται το τεταρτημόριο του κύκλου ίσο με 50, 100, 2000, 10000, 500000, 2000000 και αριθμό διεργασιών ίσο με 15).

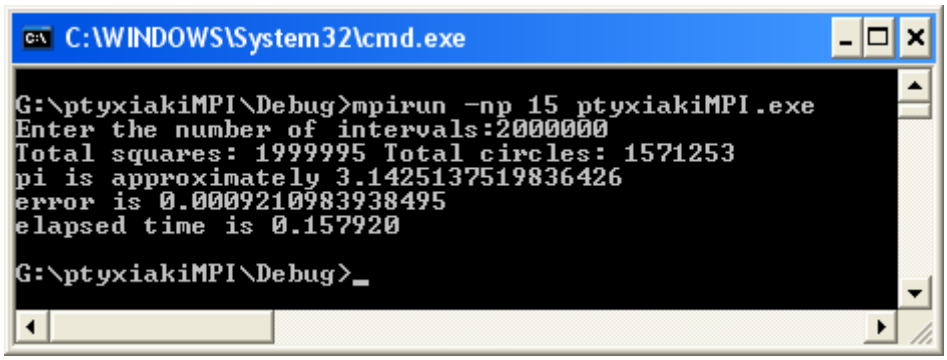












Παρατηρούμε ότι καθώς ο αριθμός των υποδιαστημάτων αυξάνεται, το σφάλμα υπολογισμού γενικά μειώνεται. Σε κάποιες περιπτώσεις, όμως, ενώ αυξάνεται ο αριθμός των υποδιαστημάτων, το σφάλμα επίσης αυξάνεται. Το γεγονός αυτό οφείλεται στο ότι η διαδικασία βασίζεται στην παραγωγή τυχαίων αριθμών, οι οποίοι δεν δίνουν πάντα το επιθυμητό αποτέλεσμα.

ΚΕΦΑΛΑΙΟ 6

6. Ο αλγόριθμος Jacobi

6.1. Αριθμητική επίλυση γραμμικών συστημάτων

Με τον όρο γραμμικά συστήματα $n \times n$ ($n = 2,3,\ldots$), εννοούμε συστήματα n εξισώσεων με n αγνώστους της μορφής:

$$\begin{aligned} a_{11}x_1 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + \ldots + a_{2n}x_n &= b_2 \\ &\ldots \\ a_{n1}x_1 + \ldots + a_{nn}x_n &= b_n \end{aligned}, \quad (6.1.1)$$

όπου x_i, $i = 1,\ldots,n$, είναι οι *άγνωστοι όροι*, $a_{i,j}$, $i,j = 1,\ldots,n$ είναι οι *συντελεστές* των αγνώστων όρων και b_i, $i = 1,\ldots,n$ είναι οι *σταθεροί* όροι. Το σύστημα (6.1.1) μπορεί να γραφεί με τη μορφή πινάκων ως εξής:

$$A\,x = B,$$

όπου

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

είναι ο ***πίνακας των συντελεστών των αγνώστων***,

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

είναι ο ***πίνακας στήλη των αγνώστων*** και

$$b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

είναι ο ***πίνακας στήλη των σταθερών όρων***.

6.2. Επαναληπτικές μέθοδοι επίλυσης συστημάτων

Από τη βασική θεωρία των μαθηματικών είναι γνωστό πως η επίλυση ενός γραμμικού συστήματος Ν εξισώσεων με Ν αγνώστους μπορεί να πραγματοποιηθεί με τη βοήθεια επαναληπτικών μεθόδων (iterative methods). Σε αυτές τις μεθόδους ξεκινούμε από μία αρχική προσέγγιση για τη λύση Χ του συστήματος και δια της επαναληπτικής εφαρμογής κάποιου αλγορίθμου προσπαθούμε να προσεγγίσουμε την πραγματική λύση. Εάν αυτή η λύση υπάρχει, οδηγούμαστε σε μια ακολουθία διαδοχικών προσεγγίσεων Χ(1), Χ(2), Χ(3)… το όριο της οποίας είναι η επιθυμητή λύση. Αυτή η επαναληπτική ακολουθία διακόπτεται όταν συμπληρωθεί ο κύκλος των επαναλήψεων που έχει προκαθορισθεί ή όταν η απόσταση ανάμεσα σε δύο διαδοχικές προσεγγίσεις γίνει μικρότερη από κάποιο επίπεδο ανοχής (tolerance).

6.3. Ο αλγόριθμος Jacobi

Ο αλγόριθμος Jacobi αποτελεί μια πολύ γνωστή μέθοδο επίλυσης γραμμικών αλγεβρικών συστημάτων με n αγνώστους. Σύμφωνα με τη μέθοδο αυτή, επιλέγεται μια αρχική προσεγγιστική λύση X^0 και μέσα από μια επαναληπτική διαδικασία, ο αλγόριθμος προσπαθεί να βρει την πραγματική λύση Χ. Αν η λύση αυτή υπάρχει, βρίσκεται ως το όριο μιας ακολουθίας διαδοχικών προσεγγίσεων X^1, X^2,..., X^m,… Αυτή η επαναληπτική διαδικασία τερματίζεται όταν ο αλγόριθμος φτάσει στην πραγματική λύση ή όταν ο προκαθορισμένος αριθμός επαναλήψεων ξεπεραστεί.

Για μια πιο μαθηματική περιγραφή, ας υποθέσουμε ότι έχουμε το παρακάτω αλγεβρικό σύστημα:

$$\begin{aligned}
&\alpha_{11}x_1 + \alpha_{12}x_2 + \ldots + \alpha_{1j}x_j + \ldots + \alpha_{1n}x_n = \beta_1 \\
&\alpha_{21}x_1 + \alpha_{22}x_2 + \ldots + \alpha_{2j}x_j + \ldots + \alpha_{2n}x_n = \beta_2 \\
&\ldots\ldots\ldots\ldots \\
&\alpha_{i1}x_1 + \alpha_{i2}x_2 + \ldots + \alpha_{ij}x_j + \ldots + \alpha_{in}x_n = \beta_i \\
&\ldots\ldots\ldots\ldots \\
&\alpha_{n1}x_1 + \alpha_{n2}x_2 + \ldots + \alpha_{nj}x_j + \ldots + \alpha_{nn}x_n = \beta_n
\end{aligned} \qquad (6.3.1)$$

Το σύστημα αυτό μπορεί να αναπαρασταθεί σαν μια εξίσωση πινάκων AX=B, όπου

$$A = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1j} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2j} & \cdots & \alpha_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{i1} & \alpha_{i2} & \cdots & \alpha_{ij} & \cdots & \alpha_{in} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{n1} & \alpha_{n2} & \cdots & \alpha_{nj} & \cdots & \alpha_{nn} \end{pmatrix}, \quad \mathbf{X} = \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_i \\ \cdots \\ x_n \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \cdots \\ \beta_i \\ \cdots \\ \beta_n \end{pmatrix}$$

είναι οι πίνακες των συντελεστών, της λύσης του συστήματος και ο πίνακας-στήλη με τους σταθερούς όρους, αντίστοιχα. Η βασική ιδέα πίσω από τον αλγόριθμο Jacobi είναι η μετατροπή του συστήματος σε ένα ισοδύναμο σύστημα της μορφής X=TX+C. Για να επιτύχουμε τη μετατροπή αυτή, πρώτα γράφουμε τον πίνακα A σαν το άθροισμα ενός διαγώνιου πίνακα D, ενός άνω τριγωνικού πίνακα L και ενός κάτω τριγωνικού πίνακα U.

$$A=\begin{pmatrix}\alpha_{11} & 0 & \cdots & 0\\ 0 & \alpha_{22} & \ddots & 0\\ \vdots & \ddots & \ddots & 0\\ 0 & \cdots & 0 & \alpha_{nn}\end{pmatrix}-\begin{pmatrix}0 & 0 & \cdots & 0\\ -\alpha_{21} & \vdots & & 0\\ \vdots & \vdots & \vdots & \vdots\\ -\alpha_{n1} & \cdots & -\alpha_{n,n-1} & 0\end{pmatrix}-\begin{pmatrix}0 & -\alpha_{12} & \cdots & -\alpha_{1n}\\ \vdots & \ddots & & 0\\ \vdots & & \vdots & -\alpha_{n-1,n}\\ 0 & \cdots & \cdots & 0\end{pmatrix}$$

Με τον τρόπο αυτό, η εξίσωση AX=B μπορεί να γραφτεί (D-L-U)X=B ή ισοδύναμα DX=(L+U)X+B. Επομένως, αν $a_{ii}\neq 0$ (i=1,2,…,n) ο αντίστροφος πίνακας D^{-1} υπάρχει και το διάνυσμα X που αποτελεί την λύση του συστήματος μπορεί να υπολογιστεί μέσω της εξίσωσης $X=D^{-1}(L+U)X+D^{-1}B$. Μέσω της τελευταίας εξίσωσης γίνεται φανερό ότι το διάνυσμα X μπορεί να υπολογιστεί αριθμητικά επιλέγοντας μια αρχική προσέγγιση X^0 και χρησιμοποιώντας στη συνέχεια την επαναληπτική εξίσωση $X^{(k)}=D^{-1}(L+U)X^{(k-1)}+D^{-1}B=TX^{(k-1)}+C$ (k=1,2,…), όπου $T=D^{-1}(L+U)$ και $C=D^{-1}B$.

Αν και η ακολουθία των διαδοχικών διανυσμάτων που παράγονται με τον τρόπο αυτό δεν συγκλίνει απαραίτητα σε κάποιο όριο, μπορεί να αποδειχτεί ότι το όριο αυτό υπάρχει αν $|\alpha_{ii}|>\Sigma_i|\alpha_{ij}|$. Αυτό το όριο θα προσεγγιστεί αν η Ευκλείδεια απόσταση μεταξύ ουο οιαδοχικών διανυσμάτων $X^{(k)}$ και $X^{(k+1)}$ γίνει μικρότερη από ένα προκαθορισμένο από τον χρήστη κατώφλι. Επειδή, όμως, είναι πιθανό ο αλγόριθμος να μην συγκλίνει σε κάποιο όριο, η επαναληπτική διαδικασία τερματίζεται όταν ο αριθμός των επαναλήψεων έχει ξεπεράσει κάποιον αριθμό που έχει προκαθορίσει ο χρήστης.

Μέσω της παραπάνω περιγραφής, γίνεται φανερό ότι ο αλγόριθμος θα αποτύχει εάν $a_{ii}=0$ για οποιοδήποτε i. Ωστόσο, αν το σύστημα έχει μοναδική λύση, οι εξισώσεις μπορεί να γραφτούν έτσι ώστε $a_{ii}\neq 0$. Στην περίπτωση αυτή, το i-οστο στοιχείο του διανύσματος $X^{(k)}$ μπορεί να υπολογιστεί από την εξίσωση:

$$x_i^{(k)}=\frac{1}{\alpha_{ii}}\{\sum_{j=1,j\neq i}^{n}(-\alpha_{ij}x_j^{(k-1)})+\beta_i\}\quad (i=1,2,\ldots,n) \tag{6.3.2}$$

Ο ψευδοκώδικας που αναπαριστά τον αλγόριθμο Jacobi παρουσιάζεται παρακάτω:
Είσοδος: (α) nxn πίνακας συντελεστών με $a_{ii}\neq 0$, (β) nx1 διάνυσμα σταθερών όρων, (γ) η αρχική προσέγγιση $X^t=X^0$, (δ) το σφάλμα ανοχής tol και ο μέγιστος αριθμός επαναλήψεων L.

1) k=1
2) όσο k≤L κάνε
   (α) για κάθε i=1,2,…,n, $x_i = -\sum_{j=1, j\neq i}^{n} (\alpha_{ij} X_j^t + \beta_i) / \alpha_{ii}$
   (β) αν $\|X - X^t\| < tol$, τότε επέστρεψε το X και σταμάτα
   (γ) k=k+1;
   (δ) $X^t = X$
3) Αν ο αριθμός των επαναλήψεων έχει φτάσει τον μέγιστο αριθμό, τερμάτισε την διαδικασία

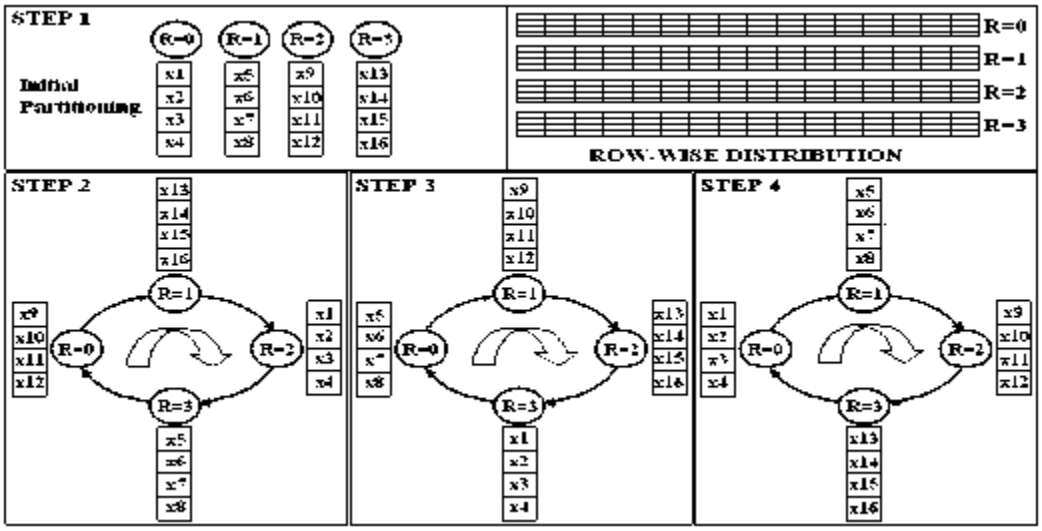
Ο παραπάνω ψευδοκώδικας περιγράφει την σειριακή λειτουργία του αλγορίθμου Jacobi όπως αυτή μπορεί να εκτελεστεί από μία CPU. Ωστόσο, ο αλγόριθμος Jacobi μπορεί εύκολα να παραλληλοποιηθεί αφού περιλαμβάνει πολλαπλασιασμούς πινάκων/διανυσμάτων που αποτελούν διαδικασίες που μπορούν να εκτελούνται παράλληλα.

6.4. Παραλληλοποίηση του αλγορίθμου Jacobi

Κατά την παραλληλοποίηση του αλγορίθμου Jacobi, υποθέτουμε για ευκολία ότι ο αριθμός p των διεργασιών διαιρεί ακριβώς την διάσταση n του nxn πίνακα A και τα διανύσματα X και B, έτσι ώστε η διανομή των δεδομένων να γίνεται ανά-γραμμή ή ανά-στήλη. Στην πρώτη περίπτωση, blocks από m=n/p γραμμές του πίνακα A αποστέλλονται στις διεργασίες του συστήματος και τα διανύσματα X και B αποστέλλονται με παρόμοιο τρόπο. Όσον αφορά την δεύτερη προσέγγιση, τα δύο διανύσματα (X και B) αποστέλλονται με τον ίδιο τρόπο ενώ ο πίνακας A χωρίζεται και αποστέλλεται όχι σε γραμμές αλλά σε στήλες (κάθε διεργασία λαμβάνει ένα σύνολο από m διαδοχικές στήλες).

6.4.1. Διανομή δεδομένων ανά-γραμμή

Κατά την διανομή δεδομένων ανά-γραμμή, μόνο ένα μέρος του διανύσματος X είναι διαθέσιμο σε κάθε διεργασία και, επομένως, ο πολλαπλασιασμός πίνακα-διανύσματος δε μπορεί να πραγματοποιηθεί άμεσα. Για τον σχηματισμό ολόκληρου του διανύσματος λύσης σε κάθε επανάληψη του αλγορίθμου η συνάρτηση MPI_Allgather μπορεί να κληθεί για να μαζευτούν τα επιμέρους διανύσματα από όλες τις διεργασίες και να συνδεθούν μεταξύ τους ώστε να σχηματιστεί ολόκληρο το διάνυσμα της λύσης. Ωστόσο, αυτή η προσέγγιση απαιτεί μεγάλο φόρτο στο κομμάτι της επικοινωνίας και δεν είναι τόσο αποδοτική όταν ο αριθμός των διεργασιών είναι μεγάλος. Μια πιο κατάλληλη τεχνική παραλληλοποίησης βασίζεται στην κυκλική μετατόπιση των δεδομένων του διανύσματος X, όπως παρουσιάζεται στο παρακάτω σχήμα (σχήμα 28).



Σχήμα 28. *Κυκλική μετατόπιση των δεδομένων κατά την διανομή ανά-γραμμή*

Από το παραπάνω σχήμα γίνεται φανερό ότι σε κάθε βήμα το μερικό διάνυσμα μετατοπίζεται προς τα επάνω και στη συνέχεια ο πολλαπλασιασμός του μερικού πίνανακα-διανύσματος πραγματοποιείται. Κατά τον τρόπο αυτό, ο πολλαπλασιαμός AX ολοκληρώνεται σε p βήματα, όπου p είναι ο αριθμός των διεργασιών. Λόγω του ότι αυτή η προσέγγιση απαιτεί κάθε διεργασία να επικοινωνεί μόνο με την πιο πάνω (για αποστολή) και την πιο κάτω (για παραλαβή) γειτονική διεργασία, η επικοινωνία έχει μικρότερο κόστος συγκριτικά με την προσέγγιση που απαιτεί την χρήση της συνάρτησης MPI_Allgather. Επιπρόσθετα, αν χρησιμοποιούνται non-blocking αιτήσεις (requests) για την αποστολή και λήψη, επιτυγχάνεται η επικάλυψη μεταξύ των υπολογισμών και της επικοινωνίας. Στην πραγματικότητα, δεδομένου ότι αυτές οι αιτήσεις (requests) δεν σημειώνουν καθυστέρηση, κάθε διεργασία μπορεί να πραγματοποιεί τους υπολογισμούς με τα δεδομένα που είναι διαθέσιμα κατά την διάρκεια των λειτουργιών μετάδοσης δεδομένων και να ελέγχει περιοδικά την κατάσταση της non-blocking επικοινωνίας χρησιμοποιώντας την συνάρτηση MPI_Test (στην περίπτωση κατά την οποία οι υπολογισμοί ολοκληρώνονται πριν την ολοκλήρωση της μετάδοσης δεδομένων, η διεργασία θα πρέπει να περιμένει, καλώντας την συνάρτηση MPI_Wait).

Σε μια μαθηματική περιγραφή, η διεργασία με τάξη R ($0 \le R \le p-1$) υπολογίζει το επιμέρους διάνυσμα $X_p=\{X_{Rm}, X_{Rm+1}, X_{Rm+2}, \ldots, X_i, \ldots, X_{(R+1)m-1}\}$. Σε κάθε ένα από τα p βήματα, μόνο ένα μέρος της τιμής κάθε στοιχείου υπολογίζεται. Πιο συγκεκριμένα, η επιμέρους τιμή ενός X_i στοιχείου $[Rm \le i \le (Rm+m-1)]$ που υπολογίζεται κατά το βήμα s $[0 \le s \le (p-1)]$ δίνεται από την εξίσωση:

$$X_{i(s)}^{(k+1)} = \sum_{j=s*m}^{(s+1)m-1} \alpha_{ij} x_j^{(k)} \qquad (6.4.1.1)$$

Όσον αφορά την τελική τιμή του στοιχείου X_i $[Rm \le i \le (R+1)m-1]$, αυτή μπορεί να υπολογιστεί αθροίζοντας όλες αυτές τις επιμέρους τιμές και, επομένως, δίνεται από την εξίσωση:

$$X_i^{(k+1)} = \sum_{s=0}^{p-1} X_{i(s)}^{(k+1)} = \sum_{s=0}^{p-1} \{ \sum_{j=s*m}^{(s+1)m-1} a_{ij} x_j^{(k)} \} \qquad (6.4.1.2)$$

Τελικά, για τον έλεγχο της συνθήκης τερματισμού

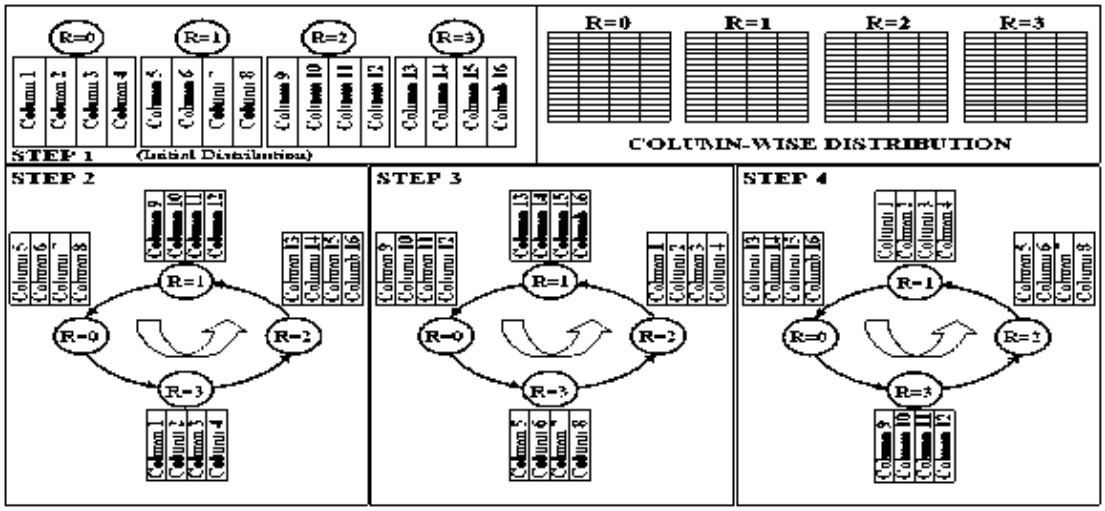
$$\left\| X^{(k+1)} - X^{(k)} \right\| = \sqrt{\sum_i (X_i^{(k+1)} - X_i^{(k)})^2} < tol \qquad (6.4.1.3)$$

απαιτείται μια καθολική λειτουργία συλλογής δεδομένων, η οποία μπορεί εύκολα να πραγματοποιηθεί με την κλήση της συνάρτησης MPI_Gather και της επιλογή μιας από τις διεργασίες (για παράδειγμα η διεργασία με τάξη R=0) ως η βασική διεργασία για τη λειτουργία αυτή.

6.4.2. Διανομή δεδομένων ανά-στήλη

Κατά την διανομή των δεδομένων ανά στήλη, κάθε διεργασία λαμβάνει ένα σύνολο από m στήλες και ένα μέρος του X διανύσματος και, επομένως, το επιμέρους αποτέλεσμα AX μπορεί να υπολογιστεί ανεξάρτητα και ταυτόχρονα με τις υπόλοιπες διεργασίες. Το γεγονός αυτό γίνεται εύκολα κατανοητό αν σκεφτούμε ότι το σύνολο m διαδοχικών στηλών σχηματίζει έναν υπο-πίνακα δύο διαστάσεων με διαστάσεις nxm, και ο πολλαπλασιασμός με το μερικό διάνυσμα διαστάσεων mx1 οδηγεί σε ένα διάνυσμα με διαστάσεις nx1. Αυτό σημαίνει ότι κάθε διεργασία υπολογίζει ένα μέρος της τιμής κάθε στοιχείου του διανύσματος X και για να υπολογιστεί η τελική τιμή, όλες αυτές οι μερικές τιμές, που υπολογίστηκαν από όλες τις διεργασίες, πρέπει να προστεθούν μεταξύ τους και το αποτέλεσμα να διαιρεθεί με την τιμή του αντίστοιχου κελιού του διανύσματος B. Ένας προφανής τρόπος για να επιτευχθεί η διαδικασία αυτή είναι να καλέσουμε την συνάρτηση MPI_Allreduce με το MPI_SUM ως όρισμα ώστε να πραγματοποιήσουμε μια συνολική αναγωγή αλλά ο πιο προτιμότερος τρόπος είναι πάλι με κυκλική μετατόπιση, η οποία στην περίπτωση αυτή πραγματοποιείται ως εξής:

Ο πίνακας των συντελεστών μετατοπίζεται προς τα αριστερά ενώ το διάνυσμα των αγνώστων μετατοπίζεται προς τα πάνω και στη συνέχεια το μερικό γινόμενο AX διαιρείται με το διάνυσμα B. Η διαδικασία αυτή πραγματοποιείται σε p βήματα. Στη συνέχεια, το συνολικό γινόμενο AX υπολογίζεται και διαιρείται από το διάνυσμα B παρέχοντας έτσι το διάνυσμα λύσης X. Η λειτουργία της κυκλικής μετατόπισης των στηλών παρουσιάζεται στο σχήμα 29 (η διαδικασία είναι παρόμοια με την κυκλική μετατόπιση που συμβαίνει κατά την διανομή δεδομένων ανά-γραμμή).



Σχήμα 29. *Κυκλική μετατόπιση των δεδομένων κατά την διανομή ανά-στήλη*

Από το παραπάνω σχήμα γίνεται φανερό ότι η διεργασία R [0≤R≤(p-1)] κατά την διάρκεια του βήματος s [0≤s≤(p-1)] κρατά τις στήλες με δείκτη (index) [(R+s)m mod n], [(R+s)m mod n]+1, …, [(R+s)m mod n]+m-1 όπου mod είναι ο τελεστής modulo. Επομένως, η μερική τιμή του στοιχείου X_i (0≤i≤n) που υπολογίζεται κατά το βήμα s δίνεται από την εξίσωση:

$$X_{i(s)}^{(k+1)} = B_i - \sum_{j=[(R+s)m \bmod n]}^{[(R+s)m \bmod n]+m-1} \alpha_{ij} x_j^{(k)} \qquad (6.4.2.1)$$

(όπου B_i είναι η τιμή του αντίστοιχου κελιού του διανύσματος B), ενώ η συνολική τιμή του στοιχείου X_i (0≤i≤n) υπολογίζεται από την εξίσωση:

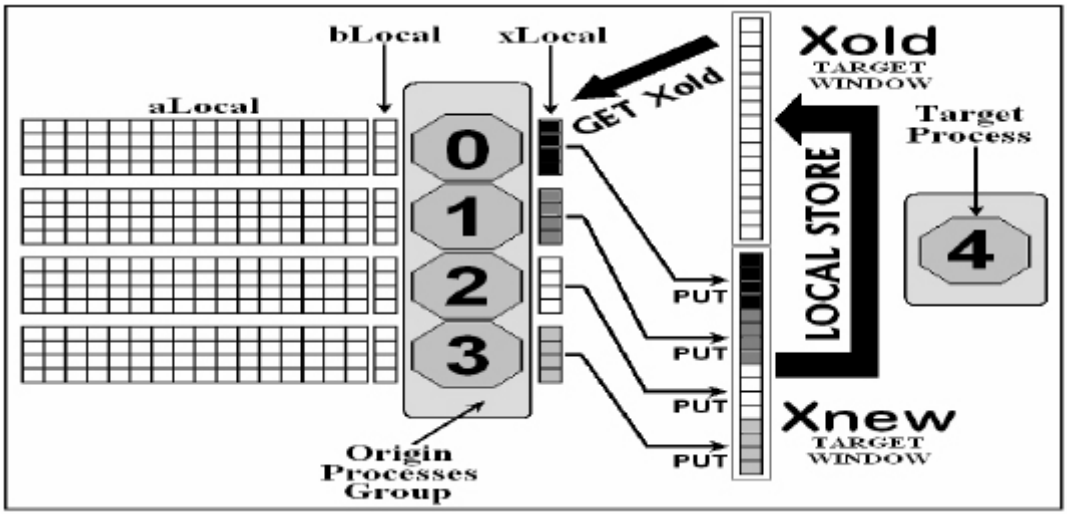
$$X_i^{(k+1)} = B_i - \sum_{s=0}^{p-1} X_{i(s)}^{(k+1)} = B_i - \sum_{s=0}^{p-1} \{ \sum_{j=[(R+s)m \bmod n]}^{[(R+s)m \bmod n]+m-1} \alpha_{ij} x_j^{(k)} \} \qquad (6.4.2.2)$$

Από την παραπάνω περιγραφή, γίνεται φανερό ότι το διάνυσμα $X^{(k+1)}$ υπολογίζεται από κάθε διεργασία και επομένως η εγκυρότητα της συνθήκης τερματισμού μπορεί να ελεγχθεί από κάποια διεργασία (π.χ. από την διεργασία με τάξη R=0). Η μέθοδος διανομής δεδομένων ανά-στήλη απαιτεί περισσότερη επικοινωνία από ότι η διανομή δεδομένων ανά-γραμμή καθώς εκτός από την μετατόπιση του διανύσματος X προς τα πάνω, ο πίνακας συντελεστών A επίσης μετατοπίζεται προς τα αριστερά. Ωστόσο, στην περίπτωση αυτή, δεν απαιτείται καθολική αναγωγή καθώς σε κάθε βήμα υπολογίζεται μια μερική τιμή για όλα τα κελιά. Επομένως, δεν υπάρχει τρόπος άμεσης σύγκρισης της αποδοτικότητας των δύο μεθόδων.

6.4.3. Μονόπλευρη υλοποίηση του αλγορίθμου Jacobi

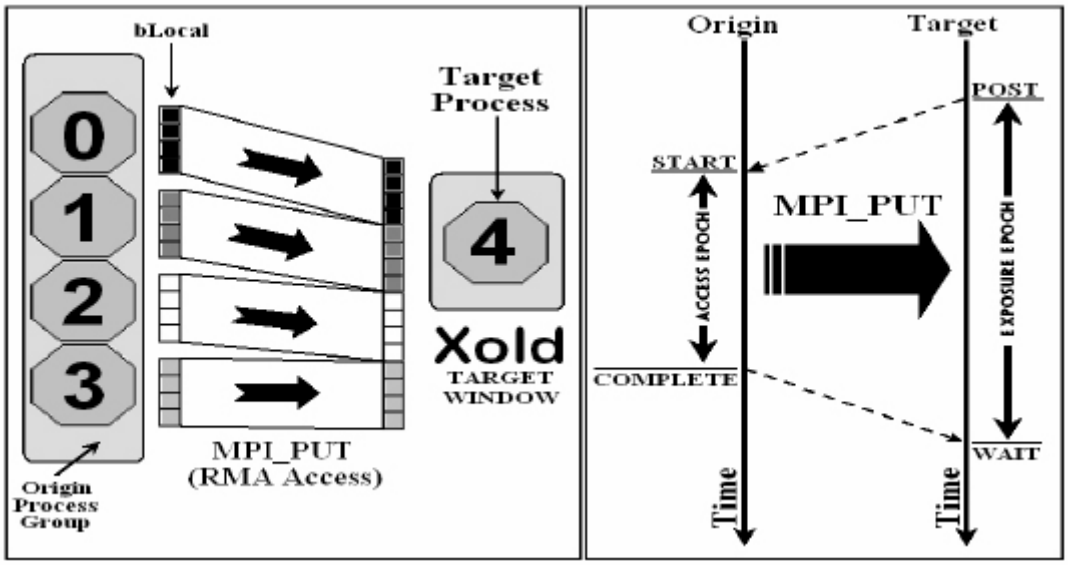
Μια εναλλακτική, αρκετά διαφορετική τεχνική παραλληλοποίησης του αλγορίθμου Jacobi μπορεί να υλοποιηθεί χρησιμοποιώντας τις συναρτήσεις μονόπλευρης επικοινωνίας που προσφέρει η βιβλιοθήκη MPI και επιτρέπουν την χρήση λειτουργιών remote memory access. Η βασική ιδέα αυτής της

προσέγγισης είναι η χρήση μιας επιπρόσθετης target διεργασίας για την αποθήκευση των διανυσμάτων $X^{(k)}$ και $X^{(k+1)}$ και τον υπολογισμό της απόστασης των τρεχουσών τιμών τους, έτσι ώστε να ελέγχεται η συνθήκη τερματισμού. Επομένως, συνολικά συμμετέχουν p+1 διεργασίες, οι πρώτες p των οποίων έχουν τάξεις {0,1,2,...,p-2} όπως προηγουμένως, ενώ η τελευταία (επιπρόσθετη) διεργασία με τάξη R=p-1 είναι η target διεργασία. Τα διανύσματα $X^{(k)}$ και $X^{(k+1)}$ αποθηκεύονται στα κατάλληλα memory windows που δημιουργούνται στη μνήμη της target διεργασίας και τα περιεχόμενά τους διαβάζονται και γράφονται από τις υπόλοιπες διεργασίες χρησιμοποιώντας τις συναρτήσεις MPI_Get και MPI_Put αντίστοιχα. Μετά τον υπολογισμό ολόκληρου του διανύσματος $X^{(k+1)}$, η απόσταση $D=||X^{(k+1)} - X^{(k)}||$ υπολογίζεται από την target διεργασία και αν είναι μικρότερη από το κατώφλι που όρισε ο χρήστης, ο αλγόριθμος τερματίζεται. Σε αντίθετη περίπτωση, το διάνυσμα $X^{(k+1)}$ αντιγράφεται στο memory window που κρατάει την παλιά τιμή του διανύσματος $X^{(k)}$. Αυτή η λειτουργία αντιγραφής πραγματοποιείται τοπικά στη μνήμη της target διεργασίας χωρίς να απαιτείται επικοινωνία με τις υπόλοιπες διεργασίες του συστήματος. Η επικοινωνία των διεργασιών στην μονόπλευρη υλοποίηση του αλγορίθμου Jacobi παρουσιάζεται στο σχήμα 30.



Σχήμα 30. *Επικοινωνία δεδομένων στο μονόπλευρο αλγόριθμο*

Ο συγχρονισμός των διεργασιών στην προσέγγιση αυτή μπορεί να πραγματοποιηθεί καλώντας τις συναρτήσεις MPI_Win_post, MPI_Win_wait, MPI_Win_start και MPI_Win_complete. Επομένως, η target διεργασία θεωρείται σαν μια ενεργή διεργασία, ενώ εναλλακτικές υλοποιήσεις μπορούν να σχεδιαστούν βασισμένες στην παθητική target διεργασία και στην χρήση των συναρτήσεων MPI_Win_lock και MPI_Win_unlock. Για τη χρήση των συναρτήσεων συγχρονισμού, δύο σύνολα από διεργασίες πρέπει να δημιουργηθούν: το σύνολο origin που περιέχει τις διεργασίες με τάξεις {0,1,2,...,p-2} και το σύνολο target που περιέχει την τελευταία target διεργασία (σχήμα 31).



Σχήμα 31. *Συγχρονισμός διεργασιών στη μονόπλευρη υλοποίηση του αλγορίθμου Jacobi*

Σύμφωνα με την παραπάνω περιγραφή, η μονόπλευρη υλοποίηση του αλγορίθμου Jacobi αποτελείται από τα εξής βήματα:

(1) Η βασική διεργασία (R=0) αρχικοποιεί τον πίνακα A και τα διανύσματα X και B και τα διανέμει σε όλες τις διεργασίες εκτός από την target χρησιμοποιώντας διανομή δεδομένων ανά-γραμμή.

(2) Η target διεργασία (R=p-1) καλεί την συνάρτηση MPI_Win_create για την δημιουργία των memory windows που θα χρησιμοποιηθούν για την αποθήκευση των διανυσμάτων $X^{(k)}$ και $X^{(k+1)}$. (Σημείωση: Καθώς αυτή είναι μια συνάρτηση για την συλλογή δεδομένων, θα κληθεί από όλες τις διεργασίες. Ωστόσο, όλες οι διεργασίες εκτός από την target θα καλέσουν την συνάρτηση με όρισμα MPI_BOTTOM για να αποτραπεί η δημιουργία του window στη μνήμη της κάθε διεργασίας).

(3) Τα σύνολα των origin και target διεργασιών δημιουργούνται και στη συνέχεια κάθε διεργασία αποθηκεύει το μερικό διάνυσμα στην κατάλληλη θέση του window μνήμης $X^{(k)}$ στη μνήμη της target διεργασίας χρησιμοποιώντας την συνάρτηση MPI_Put.

(4) Για κάθε επανάληψη k ($0 \le k \le L-1$)

- Κάθε non target διεργασία καλεί την συνάρτηση MPI_Get για να λάβει το διάνυσμα $X^{(k)}$ από το κατάλληλο window μνήμης.
- Κάθε non target διεργασία υπολογίζει το μερικό διάνυσμα $X^{(k+1)}$ και το αποθηκεύει στις κατάλληλες θέσεις μνήμης της target διεργασίας.
- Η target διεργασία υπολογίζει την απόσταση D μεταξύ των δύο αυτών διανυσμάτων και στέλνει την τιμή της σε όλες τις διεργασίες του συνόλου origin. Αν η απόσταση είναι μικρότερη από το όριο που προκαθόρισε ο χρήστης, ο αλγόριθμος τερματίζει τη λειτουργία του.

Ο συγχρονισμός των διεργασίων επιτυγχάνεται με χρήση των συναρτήσεων MPI_Win_post και MPI_Win_wait από την πλευρά των target διεργασιών και των συναρτήσεων MPI_Win_start και MPI_Win_complete από την πλευρά των origin διεργασιών.

6.5. Υλοποίηση του αλγορίθμου Jacobi με χρήση του MPI

```
#include <stdio.h>
#include <assert.h>
#include <mpi.h>
#include <math.h>

int max_iterations;

double Distance(double *X_Old, double *X_New, int n_size);

main(int argc, char** argv)
{
    /* Variables Initialization */
    int n_size, NoofRows_Bloc, NoofRows, NoofCols;
    int size, rank, Root=0;
    int irow, jrow, icol, index, Iteration, GlobalRowNo;

    double **Matrix_A, *Input_A, *Input_B, *ARecv, *BRecv;
    double *X_New, *X_Old, *Bloc_X, tmp;
    double accepted_fault;

    FILE *fp;

    /* MPI Initialisation */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Read the Input file */
    if(rank == Root)
    {

        if ((fp = fopen (argv[1], "r")) == NULL)
        {
            printf("Can't open input matrix file");
            exit(-1);
        }
        fscanf(fp, "%d %d", &NoofRows,&NoofCols);
        n_size=NoofRows;

        /* Allocate memory and read data */
        Matrix_A = (double **) malloc(n_size*sizeof(double *));

        for(irow = 0; irow < n_size; irow++)
        {
            Matrix_A[irow] = (double *) malloc(n_size * sizeof(double));
            for(icol = 0; icol < n_size; icol++)
                fscanf(fp, "%lf", &Matrix_A[irow][icol]);
        }
        fclose(fp);

        if ((fp = fopen (argv[2], "r")) == NULL)
        {
            printf("Can't open input vector file");
            exit(-1);
```

```
        }

        max_iterations = atoi(argv[3]);
        accepted_fault = atof(argv[4]);

        fscanf(fp, "%d", &NoofRows);
        n_size=NoofRows;
        Input_B  = (double *)malloc(n_size*sizeof(double));
        for (irow = 0; irow<n_size; irow++)
            fscanf(fp, "%lf",&Input_B[irow]);
        fclose(fp);

        /* Convert Matrix_A into 1-D array Input_A */
        Input_A  = (double *)malloc(n_size*n_size*sizeof(double));
        index    = 0;
        for(irow=0; irow<n_size; irow++)
        {
            for(icol=0; icol<n_size; icol++)
                Input_A[index++] = Matrix_A[irow][icol];
        }
    }

    MPI_Bcast(&NoofRows, 1, MPI_INT, Root, MPI_COMM_WORLD);
    MPI_Bcast(&NoofCols, 1, MPI_INT, Root, MPI_COMM_WORLD);
    MPI_Bcast(&max_iterations, 1, MPI_INT, Root, MPI_COMM_WORLD);
    MPI_Bcast(&accepted_fault, 1, MPI_DOUBLE, Root, MPI_COMM_WORLD);

    if(NoofRows != NoofCols)
    {
        MPI_Finalize();
        if(rank == 0)
            printf("Input Matrix Should Be Square Matrix ..... \n");
        exit(-1);
    }

    /* Broad cast the size of the matrix to all */
    MPI_Bcast(&n_size, 1, MPI_INT, Root, MPI_COMM_WORLD);

    if(n_size % size != 0)
    {
        MPI_Finalize();
        if(rank == 0)
            printf("Matrix Can not be Striped Evenly ..... \n");
        exit(-1);
    }

    NoofRows_Bloc = n_size/size;
    /* Memory of input matrix and vector on each process */
    ARecv = (double *) malloc (NoofRows_Bloc * n_size* sizeof(double));
    BRecv = (double *) malloc (NoofRows_Bloc * sizeof(double));

      /* Scatter the Input Data to all process */
    MPI_Scatter (Input_A, NoofRows_Bloc * n_size, MPI_DOUBLE, ARecv, NoofRows_Bloc * n_size,
                 MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
    MPI_Scatter (Input_B, NoofRows_Bloc, MPI_DOUBLE, BRecv, NoofRows_Bloc, MPI_DOUBLE, 0,
                    MPI_COMM_WORLD);

    X_New  = (double *) malloc (n_size * sizeof(double));
    X_Old  = (double *) malloc (n_size * sizeof(double));
    Bloc_X = (double *) malloc (NoofRows_Bloc * sizeof(double));

    /* Initailize X[i] = B[i] */
    for(irow=0; irow<NoofRows_Bloc; irow++)
        Bloc_X[irow] = BRecv[irow];

    MPI_Allgather(Bloc_X, NoofRows_Bloc, MPI_DOUBLE, X_New, NoofRows_Bloc,
                      MPI_DOUBLE, MPI_COMM_WORLD);

    Iteration = 0;
    do
    {
        for(irow=0; irow<n_size; irow++)
            X_Old[irow] = X_New[irow];

        for(irow=0; irow<NoofRows_Bloc; irow++)
        {
            GlobalRowNo = (rank * NoofRows_Bloc) + irow;
            Bloc_X[irow] = BRecv[irow];
            index = irow * n_size;

            for(icol=0; icol<GlobalRowNo; icol++)
                Bloc_X[irow] -= X_Old[icol] * ARecv[index + icol];

            for(icol=GlobalRowNo+1; icol<n_size; icol++)
                Bloc_X[irow] -= X_Old[icol] * ARecv[index + icol];

            Bloc_X[irow] = Bloc_X[irow] / ARecv[index + GlobalRowNo];
        }

        MPI_Allgather(Bloc_X, NoofRows_Bloc, MPI_DOUBLE, X_New,
                          NoofRows_Bloc, MPI_DOUBLE, MPI_COMM_WORLD);
        Iteration++;
    }
    while((Iteration < max_iterations) && (Distance(X_Old, X_New, n_size) >= accepted_fault));

    /* Output vector */
    if (rank == 0)
    {
        printf ("\n");
        printf(" ---------------------------------------- \n");
        printf("Results of Jacobi Method on processor %d: \n", rank);
        printf ("\n");

        printf("Matrix Input_A \n");
        printf ("\n");
        for (irow = 0; irow < n_size; irow++)
        {
             for (icol = 0; icol < n_size; icol++)
                 printf("%.3lf   ", Matrix_A[irow][icol]);
             printf("\n");
         }
         printf ("\n");
         printf("Matrix Input_B \n");
         printf("\n");
         for (irow = 0; irow < n_size; irow++)
             printf("%.3lf\n", Input_B[irow]);

         printf ("\n");
         printf("Solution vector \n");
         printf("Number of iterations = %d\n",Iteration);
         printf ("\n");
         for(irow = 0; irow < n_size; irow++)
             printf("%.12lf\n",X_New[irow]);
         printf(" ---------------------------------------------------- \n");
    }

    MPI_Finalize();
}

double Distance(double *X_Old, double *X_New, int n_size)
{
    int  index;
    double Sum;
    Sum = 0.0;
    for(index=0; index<n_size; index++)
        Sum += (X_New[index] - X_Old[index])*(X_New[index]-X_Old[index]);
    return(sqrt(Sum));
}
```

6.5.1. Επεξήγηση του πηγαίου κώδικα

```
#include <stdio.h>
#include <assert.h>
#include <mpi.h>
#include <math.h>

int max_iterations;
```

Στην ακέραια μεταβλητή **max_iterations** θα καταχωρηθεί ο μέγιστος αριθμός των επαναλήψεων του αλγορίθμου Jacobi.

```
double Distance(double *X_Old, double *X_New, int n_size);
```

Ορίζουμε την συνάρτηση **Distance**, η οποία υπολογίζει και επιστρέφει την Ευκλείδεια απόσταση των διανυσμάτων **X_Old** και **X_New** που έχουν μέγεθος **n_size**.

```
main(int argc, char** argv)
{
        /* Variables Initialization */
        int n_size, NoofRows_Bloc, NoofRows, NoofCols;
        int size, rank, Root=0;
        int irow, jrow, icol, index, Iteration, GlobalRowNo;

        double **Matrix_A, *Input_A, *Input_B, *ARecv, *BRecv;
        double *X_New, *X_Old, *Bloc_X, tmp;
        double accepted_fault;

        FILE *fp;
```

Η μεταβλητή *fp* είναι τύπου FILE και θα την χρησιμοποιήσουμε για να διαβάσουμε τα περιεχόμενα των αρχείων εισόδου (αρχείο που περιλαμβάνει τα δεδομένα του πίνακα Α, αρχείο που περιλαμβάνει τα δεδομένα του διανύσματος Β).

```
        /* MPI Initialisation */
        MPI_Init(&argc, &argv);
```

Πριν τη χρήση οποιασδήποτε εντολής MPI, θα πρέπει να λάβει χώρα η αρχικοποίηση του περιβάλλοντος MPI, η οποία γίνεται χρησιμοποιώντας τη συνάρτηση **MPI_Init**.

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

Η τάξη της τρέχουσας διεργασίας και το συνολικό πλήθος των διεργασιών του προεπιλεγμένου Communicator που περιγράφεται από τη σταθερά **MPI_COMM_WORLD** αποθηκεύονται στις μεταβλητές **rank** και **size** τύπου integer αντίστοιχα, καλώντας τις παραπάνω συναρτήσεις.

```
        /* Read the Input file */
        if(rank == Root)
        {
```

Ο παραπάνω έλεγχος δηλώνει ότι οι εντολές που ακολουθούν θα εκτελεστούν από την βασική διεργασία (με ID ίσο με το 0).

```
if ((fp = fopen (argv[1], "r")) == NULL)
{
```

Η εντολή **fopen (argv[1], "r")** ανοίγει το αρχείο που έχει περάσει ο χρήστης σαν πρώτο όρισμα κατά την εκτέλεση του προγράμματος, διαβάζει τα περιεχόμενά του (αυτό δηλώνεται από το όρισμα "r") και επιστρέφει τον δείκτη (pointer) τύπου FILE που αντιστοιχεί στο συγκεκριμένο αρχείο. Ο δείκτης που επιστρέφεται αποθηκεύεται στη μεταβλητή **fp**. Στην συγκεκριμένη περίπτωση, το πρώτο όρισμα του προγράμματος είναι το αρχείο που περιγράφει τον πίνακα Α (Ax=B).

```
        printf("Can't open input matrix file");
        exit(-1);
}
```

Αν κάποιο πρόβλημα εμφανιστεί κατά την κλήση της **fopen** (π.χ. το αρχείο δεν υπάρχει), η fopen επιστρέφει NULL και εκτελούνται οι παραπάνω εντολές που περιέχονται στο μπλοκ που ορίζει το "if". Δηλαδή, πρώτα απ'όλα τυπώνεται μήνυμα λάθους (μέσω της εντολής **printf** και στη συνέχεια η εκτέλεση του προγράμματος τερματίζεται (μέσω της εντολής **exit**).

```
fscanf(fp, "%d %d", &NoofRows,&NoofCols);
```

Στην πρώτη γραμμή του αρχείου που περιγράφει τον πίνακα Α, ορίζουμε τον αριθμό των γραμμών και των στηλών του πίνακα και στις επόμενες γραμμές τα στοιχεία του πίνακα Α. Μέσω της εντολής **fscanf** διαβάζουμε από το αρχείο που περιγράφει τον πίνακα Α (χρησιμοποιώντας τον δείκτη **fp** που έχουμε προηγουμένως δημιουργήσει) τα δύο πρώτα νούμερα που συναντάμε (και που για να τα αναγνωρίσει σωστά η εντολή fscanf θα πρέπει να χωρίζονται με κόμμα ή tab) και τα αποθηκεύουμε στις ακέραιες μεταβλητές **NoofRows** και **NoofCols** αντίστοιχα. Με τον τρόπο αυτό τελικά στη μεταβλητή **NoofRows** αποθηκεύεται ο αριθμός των γραμμών του πίνακα Α και στη μεταβλητή **NoofCols** ο αριθμός των στηλών του πίνακα Α.

```
n_size=NoofRows;
```

Η τιμή του n για το γραμμικό σύστημα nxn που επιθυμούμε να επιλύσουμε ισούται με τον αριθμό των γραμμών του πίνακα Α (που ισούται και με τον αριθμό των στηλών). Η τιμή αυτή αποθηκεύεται στην ακέραια μεταβλητή **n_size**.

```
/* Allocate memory and read data */
Matrix_A = (double **) malloc(n_size*sizeof(double *));
```

Μέσω της εντολής **malloc** δεσμεύεται ο απαιτούμενος χώρος μνήμης για την αποθήκευση του πίνακα Α. Ο πίνακας Α θα αποθηκευτεί στη μεταβλητή **Matrix_A** που είναι είναι πίνακας αριθμών κινητής υποδιαστολής δύο

διαστάσεων. Με την παραπάνω εντολή **malloc**, δεσμεύεται ο απαραίτητος χώρος στη μνήμη για την αποθήκευση **n_size** γραμμών στον πίνακα **Matrix_A**, όπου κάθε γραμμή θα αποτελείται από ένα διάνυσμα αριθμών κινητής υποδιαστολής μιας διάστασης.

```
for(irow = 0; irow < n_size; irow++)
{
        Matrix_A[irow] = (double *) malloc(n_size * sizeof(double));
```

Για κάθε γραμμή του **Matrix_A** δεσμεύεται ο απαραίτητος χώρος μνήμης για την αποθήκευση **n_size** αριθμών κινητής υποδιαστολής (αφού ο αριθμός των στηλών του **Matrix_A** ισούται με **n_size**).

```
for(icol = 0; icol < n_size; icol++)
        fscanf(fp, "%lf", &Matrix_A[irow][icol]);
```

Μέσω της εντολής **fscanf**, διαβάζονται ένα-ένα τα στοιχεία του πίνακα Α, από τον δείκτη σε αρχείο **fp** και αποθηκεύονται στον πίνακα δύο διαστάσεων Matrix_A.

```
}
fclose(fp);
```

Η εντολή **fclose** κλείνει το αρχείο που περιγράφει τον πίνακα Α (χρησιμοποιώντας τον δείκτη **fp** που «δείχνει» στο συγκεκριμένο αρχείο) και αποδεσμεύει τη μνήμη που είχε δεσμευτεί για το αρχείο αυτό.

```
if ((fp = fopen (argv[2], "r")) == NULL)
{
```

Η εντολή **fopen (argv[2], "r")** ανοίγει το αρχείο που έχει περάσει ο χρήστης σαν δεύτερο όρισμα κατά την εκτέλεση του προγράμματος, διαβάζει τα περιεχόμενά του (αυτό δηλώνεται από το όρισμα "r") και επιστρέφει τον δείκτη (pointer) τύπου FILE που αντιστοιχεί στο συγκεκριμένο αρχείο. Ο δείκτης που επιστρέφεται αποθηκεύεται στη μεταβλητή **fp**. Στην συγκεκριμένη περίπτωση, το δεύτερο όρισμα του προγράμματος είναι το αρχείο που περιγράφει τον πίνακα (μιας διάστασης) Β (Ax=B).

```
        printf("Can't open input vector file");
        exit(-1);
}
```

Αν κάποιο πρόβλημα εμφανιστεί κατά την κλήση της **fopen** (π.χ. το αρχείο δεν υπάρχει), η fopen επιστρέφει NULL και εκτελούνται οι παραπάνω εντολές που περιέχονται στο μπλοκ που ορίζει το "if". Δηλαδή, πρώτα απ'όλα τυπώνεται μήνυμα λάθους (μέσω της εντολής **printf** και στη

συνέχεια η εκτέλεση του προγράμματος τερματίζεται (μέσω της εντολής **exit**).

```
max_iterations = atoi(argv[3]);
```

Το τρίτο όρισμα που δέχεται το πρόγραμμα είναι ο αριθμός των επαναλήψεων που θα εκτελέσει ο αλγόριθμος Jacobi, Αφού πρώτα το τρίτο όρισμα μετατραπεί σε ακέραιο αριθμό (μέσω της εντολής **atoi(argv[3])**), αποθηκεύεται στην ακέραια μεταβλητή **max_iterations**.

```
accepted_fault = atof(argv[4]);
```

Το τέταρτο όρισμα που δέχεται το πρόγραμμα είναι το σφάλμα ανοχής για τον αλγόριθμο Jacobi, Αφού πρώτα το τέταρτο όρισμα μετατραπεί σε αριθμό κινητής υποδιαστολής (μέσω της εντολής **atof(argv[4])**), αποθηκεύεται στη μεταβλητή κινητής υποδιαστολής **accepted_fault**.

```
fscanf(fp, "%d", &NoofRows);
```

Στην πρώτη γραμμή του αρχείου που περιγράφει τον πίνακα Β, ορίζουμε τον αριθμό των στοιχείων του πίνακα (ο οποίος είναι ίσος και με τον αριθμό των στηλών, αφού ο πίνακας είναι 1xn). Μέσω της εντολής **fscanf** διαβάζουμε από το αρχείο που περιγράφει τον πίνακα Β (χρησιμοποιώντας τον δείκτη **fp**) το πρώτο νούμερο που συναντάμε και το αποθηκεύουμε στην ακέραια μεταβλητή **NoofRows**. Με τον τρόπο αυτό τελικά στη μεταβλητή **NoofRows** αποθηκεύεται ο αριθμός των στοιχείων του πίνακα Β.

```
n_size=NoofRows;
```

Η τιμή του n για το γραμμικό σύστημα nxn που επιθυμούμε να επιλύσουμε ισούται με τον αριθμό των στοιχείων του πίνακα Β.

```
Input_B  = (double *)malloc(n_size*sizeof(double));
```

Μέσω της εντολής **malloc** δεσμεύεται ο απαιτούμενος χώρος μνήμης για την αποθήκευση του πίνακα Β. Ο πίνακας Β θα αποθηκευτεί στη μεταβλητή **Input_B** που είναι είναι πίνακας αριθμών κινητής υποδιαστολής μιας διάστασης. Με την παραπάνω εντολή **malloc**, δεσμεύεται ο απαραίτητος χώρος στη μνήμη για την αποθήκευση **n_size** αριθμών κινητής υποδιαστολής στον πίνακα **Input_B**.

```
for (irow = 0; irow<n_size; irow++)
        fscanf(fp, "%lf",&Input_B[irow]);
```

Μέσω της εντολής **fscanf**, διαβάζονται ένα-ένα τα στοιχεία του πίνακα Β, από τον δείκτη σε αρχείο **fp** και αποθηκεύονται στον πίνακα **Input_B**.

```
fclose(fp);
```

Η εντολή **fclose** κλείνει το αρχείο που περιγράφει τον πίνακα Β (χρησιμοποιώντας τον δείκτη **fp** που «δείχνει» στο συγκεκριμένο αρχείο) και αποδεσμεύει τη μνήμη που είχε δεσμευτεί για το αρχείο αυτό.

Στη συνέχεια θα μετατρέψουμε τον πίνακα δύο διαστάσεων **Matrix_A** σε έναν πίνακα μιας διάστασης (**Input_A**).

```
/* Convert Matrix_A into 1-D array Input_A */
Input_A  = (double *)malloc(n_size*n_size*sizeof(double));
```

Μέσω της εντολής **malloc** δεσμεύεται ο απαιτούμενος χώρος μνήμης για τον πίνακα **Input_A**. Στον πίνακα **Input_A** πρόκειται να αποθηκευτούν n_size*n_size αριθμοί κινητής υποδιαστολής (αφού το σύνολο των στοιχείων του πίνακα **Matrix_**Α ισούται με n_size*n_size).

```
index = 0;
```

Η ακέραια μεταβλητή **index** χρησιμοποιείται ως μετρητής.

```
for(irow=0; irow<n_size; irow++)
{
        for(icol=0; icol<n_size; icol++)
                Input_A[index++] = Matrix_A[irow][icol];
}
```

Οι παραπάνω δύο βρόγχοι **for** διατρέχουν όλα τα στοιχεία του πίνακα δύο διαστάσεων **Matrix_A** και τα αποθηκεύουν στον πίνακα μιας διάστασης **Input_A**.

```
}

MPI_Bcast(&NoofRows, 1, MPI_INT, Root, MPI_COMM_WORLD);
MPI_Bcast(&NoofCols, 1, MPI_INT, Root, MPI_COMM_WORLD);
MPI_Bcast(&max_iterations, 1, MPI_INT, Root, MPI_COMM_WORLD);
MPI_Bcast(&accepted_fault, 1, MPI_DOUBLE, Root, MPI_COMM_WORLD);
```

Μέσω των παραπάνω τεσσάρων συναρτήσεων **MPI_Bcast**, η βασική διεργασία (**Root**) στέλνει τις τιμές των ακεραίων μεταβλητών **NoofRows, NoofCols** και **max_iterations** καθώς επίσης και την τιμή της μεταβλητής κινητής υποδιαστολής **accepted_fault** σε όλες τις διεργασίες.

```
if(NoofRows != NoofCols)
{
        MPI_Finalize();
        if(rank == 0)
                printf("Input Matrix Should Be Square Matrix ..... \n");
        exit(-1);
}
```

Σε περίπτωση που οι δύο αριθμοί **NoofRows** και **NoofCols** δεν είναι ίσοι μεταξύ τους, αυτό σημαίνει ότι τα δεδομένα εισόδου δεν είναι σωστά και δεν έχουμε σύστημα εξισώσεων nxn αλλά nxm (με n!=m), οπότε το πρόγραμμα θα πρέπει να τυπώσει μήνυμα λάθους και να τερματιστεί.

Με την κλήση της **MPI_Finalize** απελευθερώνεται η μνήμη που δεσμεύεται από τις δομές δεδομένων του προτύπου MPI και γενικότερα τερματίζεται η λειτουργία του. Η βασική διεργασία (με ID ίσο με 0) τυπώνει μήνυμα λάθους μέσω της εντολής **printf** και στη συνέχεια η εντολή **exit(-1)** τερματίζει τη λειτουργία του προγράμματος.

```
/* Broad cast the size of the matrix to all */
MPI_Bcast(&n_size, 1, MPI_INT, Root, MPI_COMM_WORLD);
```

Μέσω της παραπάνω συνάρτησης **MPI_Bcast**, η βασική διεργασία (**Root**) στέλνει την τιμή της ακέραιας μεταβλητής **n_size** σε όλες τις διεργασίες.

```
if(n_size % size != 0)
{
        MPI_Finalize();
        if(rank == 0)
                printf("Matrix Can not be Striped Evenly ..... \n");
        exit(-1);
}
```

Σε περίπτωση που ο αριθμός **n_size** δεν διαιρεί ακριβώς το πλήθος των διεργασιών (**size**), τερματίζεται η λειτουργία του προτύπου MPI (**MPI_Finalize**), τυπώνεται μήνυμα λάθους (**printf**) από την βασική διεργασία και τερματίζεται η λειτουργία του προγράμματος (**exit(-1)**).

```
NoofRows_Bloc = n_size/size;
```

Κάθε μία από τις διεργασίες θα υπολογίσει ένα ποσοστό του διανύσματος Χ που αποτελεί και τη λύση του προβλήματος. Κάθε διεργασία θα παραλάβει ένα τμήμα του πίνακα Α (**ARecv**) με διαστάσεις **NoofRows_Bloc*n_size** και **NoofRows_Bloc** στοιχεία του διανύσματος Β (**BRecv**), με την τιμή της μεταβλητής **NoofRows_Bloc** να είναι ίση με **n_size/size**.

```
/* Memory of input matrix and vector on each process */
ARecv = (double *) malloc (NoofRows_Bloc * n_size* sizeof(double));
BRecv = (double *) malloc (NoofRows_Bloc * sizeof(double));
```

Μέσω της συνάρτησης **malloc** δεσμεύεται ο απαραίτητος χώρος μνήμης για τους πίνακες **ARecv** και **BRecv**.

```
 /* Scatter the Input Data to all process */
MPI_Scatter (Input_A, NoofRows_Bloc * n_size, MPI_DOUBLE, ARecv,
NoofRows_Bloc * n_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

MPI_Scatter (Input_B, NoofRows_Bloc, MPI_DOUBLE, BRecv,
NoofRows_Bloc, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Η συνάρτηση **MPI_Scatter** επιτρέπει τη διασπορά των δεδομένων ενός μηνύματος σε όλες τις διεργασίες της εφαρμογής. Η διασπορά, ως

λειτουργία, χαρακτηρίζεται από το διαχωρισμό ενός μηνύματος σε μικρότερα κομμάτια, κάθε ένα εκ των οποίων αποστέλλεται και σε μια διαφορετική διεργασία.

Μέσω των παραπάνω δύο συναρτήσεων **MPI_Scatter** η βασική διεργασία (με ID ίσο με 0) αποστέλλει τους πίνακες **Input_A** και **Input_B** σε όλες τις διεργασίες και οι πίνακες αυτοί αποθηκεύονται στις θέσεις μνήμης που ορίζουν τα **ARecv** και **BRecv**.

```
X_New  = (double *) malloc (n_size * sizeof(double));
X_Old  = (double *) malloc (n_size * sizeof(double));
Bloc_X = (double *) malloc (NoofRows_Bloc * sizeof(double));
```

Μέσω της συνάρτησης **malloc** δεσμεύεται ο απαραίτητος χώρος μνήμης για τους μονοδιάστατους πίνακες **X_New**, **X_Old** και **Bloc_X**. Ο πίνακας **X_New** χρησιμοποιείται για την αποθήκευση των τιμών των μεταβλητών x_i (0 < i <= n_size) όπως αυτές διαμορφώνονται στην τρέχουσα επανάληψη του αλγορίθμου Jacobi. Ο πίνακας **X_Old** χρησιμοποιείται για την αποθήκευση των τιμών των μεταβλητών x_i (0 < i <= n_size) κατά την προηγούμενη επανάληψη και ο πίνακας **Bloc_X** χρησιμοποιείται για την αποθήκευση των τιμών των μεταβλητών x_i (μέρος της λύσης – **NoofRows** στοιχεία από το σύνολο των **n_size** στοιχείων) που υπολογίζει κάθε διεργασία στην τρέχουσα επανάληχη.

```
/* Initailize X[i] = B[i] */
for(irow=0; irow<NoofRows_Bloc; irow++)
        Bloc_X[irow] = BRecv[irow];
```

Μέσω του παραπάνω βρόγχου **for** αρχικοποιείται το διάνυσμα/μονοδιάστατος πίνακας **Bloc_X,** τις τιμές του οποίου καλείται να υπολογίσει η τρέχουσα διεργασία, με τις τιμές του διανύσματος **BRecv** που έχουν αποσταλλεί στην συγκεκριμένη διεργασία.

```
MPI_Allgather(Bloc_X, NoofRows_Bloc, MPI_DOUBLE, X_New,
NoofRows_Bloc, MPI_DOUBLE, MPI_COMM_WORLD);
```

Η συνάρτηση **MPI_Allgather** επιτρέπει την παραλαβή του αποτελέσματος από όλες τις διεργασίες του συστήματος. Η χρήση της συνάρτησης **MPI_Allgather,** όπως παρουσιάζεται παραπάνω, δηλώνει ότι όλες οι διεργασίες θα παραλάβουν το διάνυσμα **Bloc_X** όπως αυτό υπολογίστηκε από κάθε διεργασία ξεχωριστά και θα το αποθηκεύσουν στο διάνυσμα **X_New**.

```
Iteration = 0;
```

Η ακέραια μεταβλητή **Iteration** αποτελεί έναν μετρητή των επαναλήψεων του αλγορίθμου Jacobi.

```
do
{
        for(irow=0; irow<n_size; irow++)
                X_Old[irow] = X_New[irow];
```

Σε κάθε επανάληψη του αλγορίθμου Jacobi το διάνυσμα **X_Old** αρχικοποιείται με τις τιμές του διανύσματος **X_New** (που είναι το αποτέλεσμα της προηγούμενης επανάληψης).

```
for(irow=0; irow<NoofRows_Bloc; irow++)
{
```

Για τον υπολογισμό της τιμής κάθε στοιχείου x_i που ανήκει στο διάνυσμα **Bloc_X** που υπολογίζει κάθε διεργασία σε μια επανάληψη, εκτελούνται οι εντολές που εμπεριέχονται στον βρόγχο **for**.

Οι τιμές του διανύσματος **Bloc_X** υπολογίζονται βάσει του τύπου

$$x_i^{(k+1)} = \frac{1}{\alpha_{ii}}(\beta_i - \sum_{j \neq i} \alpha_{ij} x_j^{(k)})$$

όπως παρουσιάστηκε προηγουμένως στην θεωρητική επεξήγηση του αλγορίθμου Jacobi.

```
GlobalRowNo = (rank * NoofRows_Bloc) + irow;
```

Αρχικά υπολογίζεται η τιμή της ακέραιας μεταβλητής **GlobalRowNo** που δηλώνει την θέση της τρέχουσας γραμμής στο διάνυσμα **X_Old**.

```
Bloc_X[irow] = BRecv[irow];
```

Η τιμές του διανύσματος **Bloc_X** αρχικοποιούνται με τις τιμές του διανύσματος **BRecv**.

```
index = irow * n_size;
```

Υπολογίζεται η τιμή της ακέραιας μεταβλητής **index**, που δηλώνει την θέση της τρέχουσας γραμμής στο διάνυσμα **ARecv**.

```
for(icol=0; icol<GlobalRowNo; icol++)
        Bloc_X[irow] -= X_Old[icol] * ARecv[index + icol];

for(icol=GlobalRowNo+1; icol<n_size; icol++)
        Bloc_X[irow] -= X_Old[icol] * ARecv[index + icol];
```

Οι παραπάνω δύο βρόγχοι **for** είναι αναγκαίοι για να αποκλείσουμε από τους υπολογισμούς την περίπτωση όπου το **icol** ισούται με το **GlobalRowNo** (βλέπε j≠i στον τύπο (6.3.2))

```
        Bloc_X[irow] = Bloc_X[irow] / ARecv[index +
        GlobalRowNo];
}

MPI_Allgather(Bloc_X, NoofRows_Bloc, MPI_DOUBLE, X_New,
        NoofRows_Bloc, MPI_DOUBLE, MPI_COMM_WORLD);
```

Η χρήση της συνάρτησης **MPI_Allgather,** όπως παρουσιάζεται παραπάνω, δηλώνει ότι όλες οι διεργασίες θα παραλάβουν το διάνυσμα **Bloc_X** όπως αυτό υπολογίστηκε από κάθε διεργασία ξεχωριστά και θα το αποθηκεύσουν στο διάνυσμα **X_New**.

```
Iteration++;
```

Η παραπάνω εντολή αυξάνει την τιμή της ακέραιας μεταβλητής **Iteration** (μετρητής επαναλήψεων) κατά 1.

```
}
while((Iteration < max_iterations) && (Distance(X_Old, X_New, n_size)
>= accepted_fault));
```

Η διαδικασία που περιγράφεται στον βρόγχο **do** θα εκτελείται όσο ισχύει η συνθήκη του **while**, δηλαδή όσο ο αριθμός των επαναλήψεων δεν έχει φτάσει τον μέγιστο αριθμό επαναλήψεων (**max_iterations** - όπως αυτός ορίστηκε από τον χρήστη) και το σφάλμα είναι μεγαλύτερο ή ίσο με το αποδεκτό σφάλμα (**accepted_fault** – όπως αυτό επίσης ορίστηκε από τον χρήστη).

```
/* Output vector */

if (rank == 0)
{
```

Η βασική διεργασία τελικά θα τυπώσει στην οθόνη τα αποτελέσματα του αλγορίθμου Jacobi.

```
printf ("\n");
printf(" ---------------------------------------- \n");
printf("Results of Jacobi Method on processor %d: \n", rank);
printf ("\n");

printf("Matrix Input_A \n");
printf ("\n");
for (irow = 0; irow < n_size; irow++)
{
        for (icol = 0; icol < n_size; icol++)
                printf("%.3lf  ", Matrix_A[irow][icol]);
        printf("\n");
}
printf ("\n");
printf("Matrix Input_B \n");
printf("\n");
for (irow = 0; irow < n_size; irow++)
        printf("%.3lf\n", Input_B[irow]);
```

Αρχικά τυπώνεται στην οθόνη ο αρχικός πίνακας Α και το διάνυσμα Β και στη συνέχεια τυπώνεται το διάνυσμα **X_New** που περιέχει τη λύση του γραμμικού συστήματος, όπως αυτή υπολογίστηκε μέσω του αλγορίθμου Jacobi.

```
            printf ("\n");
            printf("Solution vector \n");
            printf("Number of iterations = %d\n",Iteration);
            printf ("\n");
            for(irow = 0; irow < n_size; irow++)
                  printf("%.12lf\n",X_New[irow]);
            printf(" ------------------------------------------------- \n");
      }

      MPI_Finalize();
```

Με την κλήση της **MPI_Finalize** απελευθερώνεται η μνήμη που δεσμεύεται από τις δομές δεδομένων του προτύπου MPI και γενικότερα τερματίζεται η λειτουργία του.

}

Η συνάρτηση **Distance** υπολογίζει και επιστρέφει την Ευκλείδεια απόσταση των διανυσμάτων **X_Old** και **X_New** που έχουν μέγεθος **n_size**.

<u>Σημείωση:</u> Η Ευκλείδεια απόσταση μεταξύ δύο διανυσμάτων $y=(y_1, y_2, \ldots, y_p)$ *και* $x=(x_1, x_2, \ldots, x_p)$ *ισούται με :*

$$d(x,y) = \sqrt{\sum_{i=1}^{p} (x_i - y_i)^2}$$

```
double Distance(double *X_Old, double *X_New, int n_size)
{
      int  index;
      double Sum;

      Sum = 0.0;
      for(index=0; index<n_size; index++)
            Sum += (X_New[index] - X_Old[index])*(X_New[index]-
X_Old[index]);

      return(sqrt(Sum));
```

Η συνάρτηση **sqrt** επιστρέφει την τετραγωνική ρίζα ενός αριθμού, οπότε τελικά η συνάρτηση **Distance** θα επιστρέψει την τετραγωνική ρίζα του αριθμού κινητής υποδιαστολής **Sum**.

}

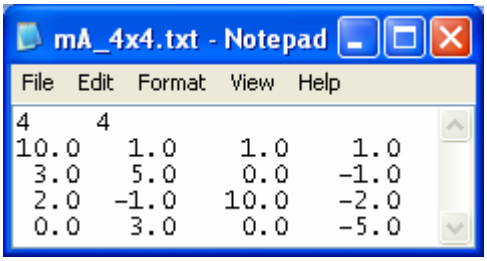
6.5.2. Παραδείγματα εκτέλεσης του προγράμματος

Παρακάτω παρουσιάζονται κάποια ενδεικτικά παραδείγματα εκτέλεσης του προγράμματος.

Α) Έστω το παρακάτω γραμμικό σύστημα 4x4 :

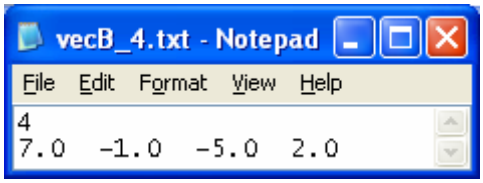
$$\begin{aligned} 10x_1 + x_2 + x_3 + x_4 &= 7 \\ 3x_1 + 5x_2 + 0x_3 - x_4 &= -1 \\ 2x_1 - x_2 + 10x_3 - 2x_4 &= -5 \\ 0x_1 + 3x_2 + 0x_3 - 5x_4 &= 2 \end{aligned}$$

Ο πίνακας Α για το συγκεκριμένο σύστημα είναι:



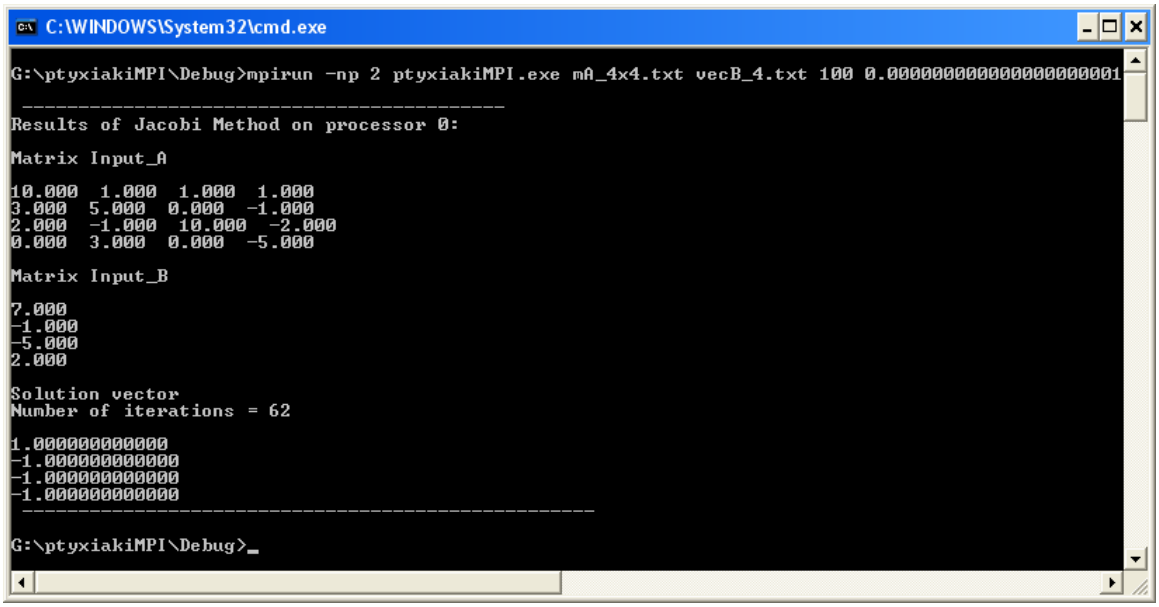
Στην πρώτη γραμμή του αρχείου ορίζεται ο αριθμός των γραμμών και στηλών του πίνακα.

Το διάνυσμα Β για το συγκεκριμένο σύστημα είναι :



Στην πρώτη γραμμή του αρχείου ορίζεται ο αριθμός των στοιχείων του διανύσματος.

Παρακάτω παρουσιάζεται το αποτέλεσμα του προγράμματος για το συγκεκριμένο γραμμικό σύστημα (αριθμός επαναλήψεων = 100, αποδεκτό σφάλμα = 0.00000000000000000001) :

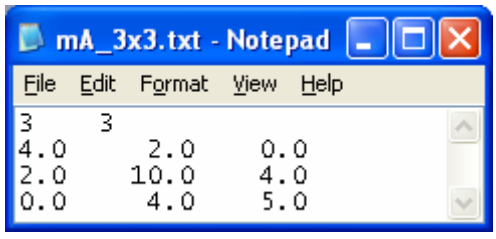


Όπως φαίνεται στην παραπάνω εικόνα, η λύση του γραμμικού συστήματος (όπως υπολογίστηκε από 2 διεργασίες) είναι το διάνυσμα X=(1,-1,-1,-1) και προσεγγίζεται μετά από 62 επαναλήψεις.

Β) Έστω το παρακάτω γραμμικό σύστημα 3x3 :

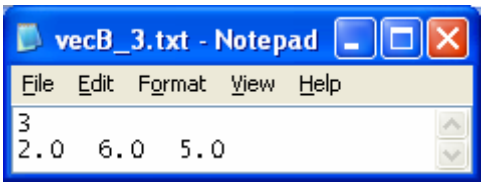
$$\begin{aligned} 4x_1 + 2x_2 &= 2 \\ 2x_1 + 10x_2 + 4x_3 &= 6 \\ 4x_2 + 5x_3 &= 5 \end{aligned}$$

Ο πίνακας Α για το συγκεκριμένο σύστημα είναι:



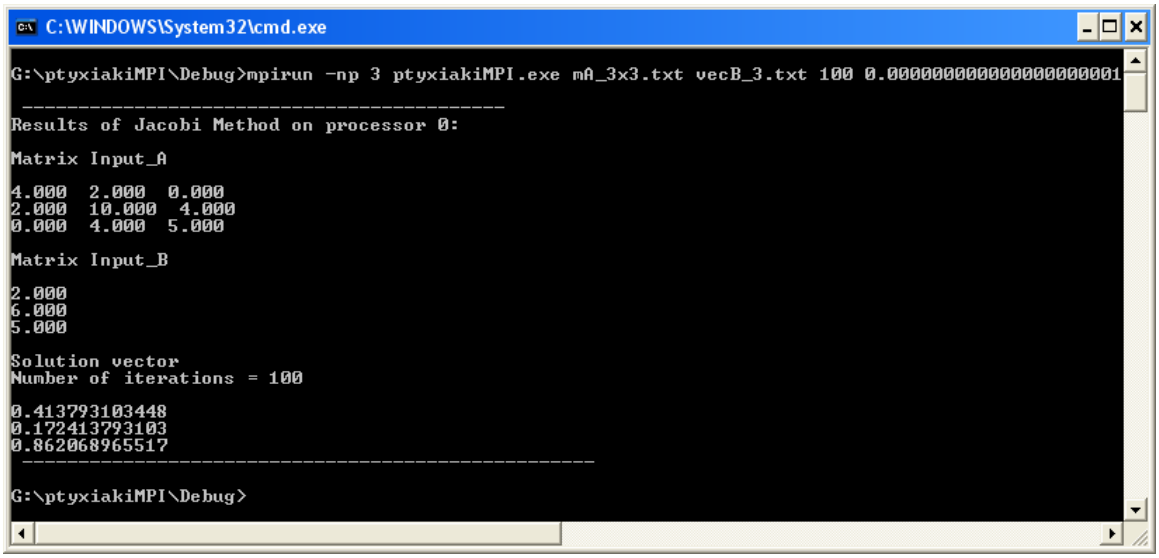
Στην πρώτη γραμμή του αρχείου ορίζεται ο αριθμός των γραμμών και στηλών του πίνακα.

Το διάνυσμα B για το συγκεκριμένο σύστημα είναι :



Στην πρώτη γραμμή του αρχείου ορίζεται ο αριθμός των στοιχείων του διανύσματος.

Παρακάτω παρουσιάζεται το αποτέλεσμα του προγράμματος για το συγκεκριμένο γραμμικό σύστημα (αριθμός επαναλήψεων = 100, αποδεκτό σφάλμα = 0.00000000000000000001) :



Όπως φαίνεται στην παραπάνω εικόνα, η λύση του γραμμικού συστήματος (όπως υπολογίστηκε από 3 διεργασίες) είναι το διάνυσμα X=(0.41379, 0.17241, 0.86206) και προσεγγίζεται μετά από 100 επαναλήψεις (100 είναι ο μέγιστος αριθμός επαναλήψεων όπως ορίστηκε κατά την εκτέλεση).

ΚΕΦΑΛΑΙΟ 7

7. Ο αλγόριθμος του Fox

Είναι γνωστό πως κατά τον πολλαπλασιασμό δύο πινάκων Α, Β, εάν οι διαστάσεις αυτών των πινάκων είναι MxN και NxL, τότε οι διαστάσεις του πίνακα γινομένου C θα είναι MxL ενώ το κάθε στοιχείο αυτού του πίνακα, C_{ij}, (που ανήκει στη γραμμή i και στη στήλη j) θα δίδεται από τη σχέση

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \quad (7.1)$$

όπου φυσικά χρησιμοποιούμε το συμβολισμό $A=\{A_{ij}\}$ και $B=\{B_{ij}\}$.

Για να εκτελεστεί ο παραπάνω πολλαπλασιασμός από ένα πλήθος διεργασιών με χρήση του MPI θα πρέπει:

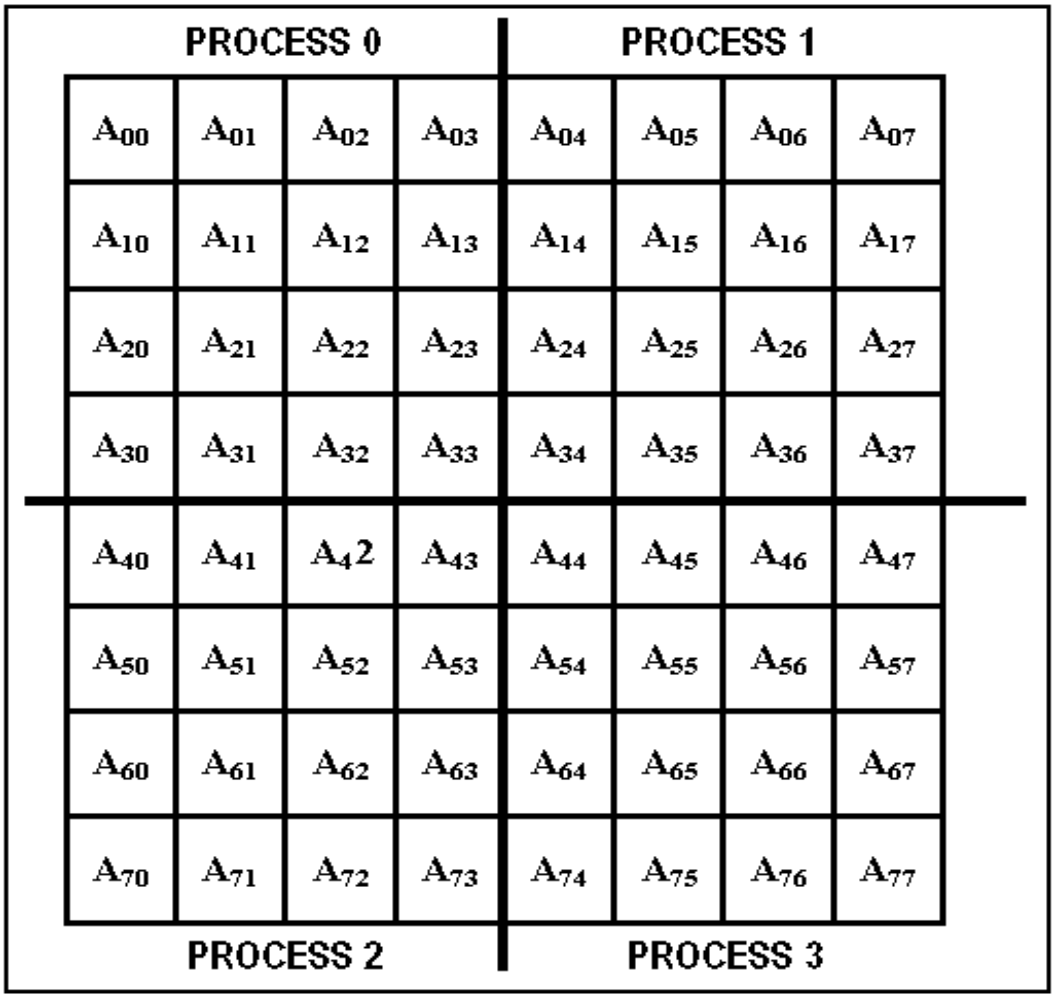
- να αποσταλούν οι γραμμές του πίνακα Α σε όλες τις διεργασίες του συστήματος, και
- να εκτελεστεί ο υπολογισμός των αντίστοιχων γραμμών του πίνακα γινομένου από την κάθε μια από αυτές

Ο υπολογισμός της τιμής του στοιχείου C_{ij} του πίνακα C από κάποια διεργασία, προϋποθέτει τη γνώση των N στοιχείων της υπ᾽αριθμόν i γραμμής του πίνακα Α, καθώς και τη γνώση των N στοιχείων της υπ’ αριθμόν j στήλης του πίνακα Β. Επομένως, ο υπολογισμός όλων των στοιχείων κάποιας γραμμής του πίνακα γινομένου, απαιτεί τη γνώση όλων των στοιχείων της αντίστοιχης γραμμής του πίνακα Α, καθώς και τη γνώση του πίνακα Β στο σύνολό του, ο οποίος επομένως θα πρέπει να αποσταλεί σε όλες τις διεργασίες του συστήματος. Η διαδικασία αυτή πραγματοποιείται με τη βοήθεια της συνάρτησης **MPI_Bcast** η οποία χρησιμοποιείται για την εκπομπή δεδομένων από τη διεργασία ρίζα προς τις υπόλοιπες διεργασίες.

Ωστόσο το βασικό μειονέκτημα που χαρακτηρίζει την προσέγγιση αυτή είναι το υψηλό υπολογιστικό κόστος, το οποίο προέρχεται από την ανταλλαγή ενός πολύ μεγάλου όγκου δεδομένων ανάμεσα στις διεργασίες εφαρμογής. Για παράδειγμα, αν υποθέσουμε πως οι πίνακες Α, Β και C είναι τετραγωνικοί πίνακες τάξεως N – δηλαδή οι διαστάσεις τους είναι NxN – ενώ το πλήθος των διεργασιών του συστήματος έχει επίσης την ίδια τιμή, αυτό σημαίνει πως η κάθε διεργασία θα υπολογίσει μια και μοναδική γραμμή του πίνακα γινομένου. Στη συνέχεια, εάν υπολογίσουμε το στοιχείο C_{ij} χρησιμοποιώντας την παραπάνω μέθοδο, θα πρέπει το σύνολο των στοιχείων της στήλης του πίνακα Β που χρησιμοποιείται σε κάθε περίπτωση να είναι γνωστό στη διεργασία που χρησιμοποιεί αυτή τη στήλη με κάποια από τις γραμμές του πίνακα Α. Το γεγονός αυτό ισχύει για κάθε γραμμή του αρχικού πίνακα, επομένως μπορούμε να διαπιστώσουμε πως ο συγκεκριμένος αλγόριθμος υπολογισμού χαρακτηρίζεται από πάρα πολύ υψηλό υπολογιστικό κόστος. Η διαπίστωση αυτή ισχύει και για τις περιπτώσεις εκείνες κατά τις οποίες η διασπορά των στοιχείων ενός πίνακα δεν γίνεται κατά γραμμές αλλά κατά στήλες.

Μια εναλλακτική προσέγγιση που μπορούμε να ακολουθήσουμε, είναι να αφήσουμε τελείως την ιδέα της διασποράς των γραμμών ή στηλών ενός πίνακα στις διεργασίες του συστήματος, και να θεωρήσουμε τις διεργασίες

αυτές διατεταγμένες στους κόμβους ενός δυσδιάστατου πλέγματος. Με την διάταξη αυτή οι διεργασίες ορίζουν μια καρτεσιανή τυπολογία. Στην περίπτωση αυτή η κάθε διεργασία δεν παραλαμβάνει μια ή περισσότερες γραμμές ή στήλες του αρχικού πίνακα, αλλά ένα τμήμα του ανωτέρω πίνακα, δηλαδή ένα υπο-πίνακα. Στο παρακάτω σχήμα (σχήμα 32) παρουσιάζεται η κατάσταση αυτή για ένα σύνολο τεσσάρων διεργασιών οι οποίες βρίσκονται διατεταγμένες στους κόμβους ενός δυσδιάστατου πλέγματος με διαστάσεις 2x2 και για ένα τετραγωνικό πίνακα με διαστάσεις 8x8.

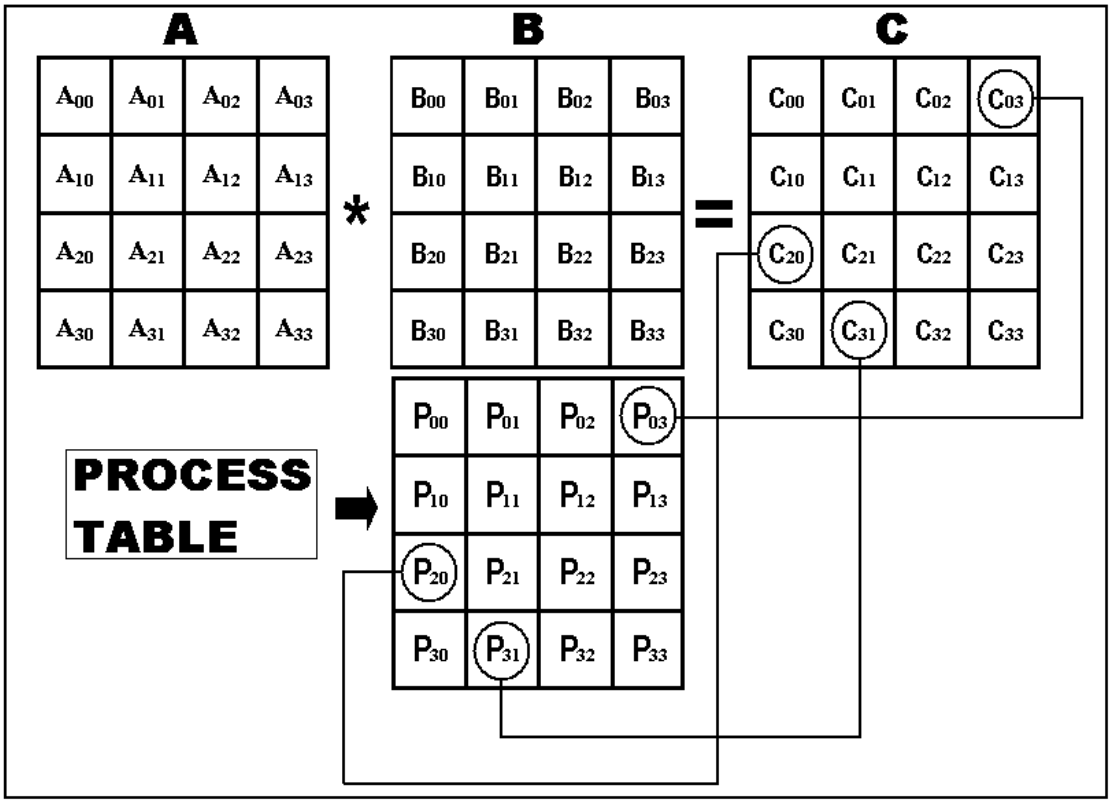


Σχήμα 32. *Κατανομή των στοιχείων δυσδιάστατου πίνακα στις διεργασίες καρτεσιανής τοπολογίας*

Από το παραπάνω σχήμα διαπιστώνουμε πως η κάθε διεργασία παραλαμβάνει μόνο ένα τμήμα του αρχικού πίνακα που στην προκειμένη περίπτωση έχει διαστάσεις 4x4. Κάθε διεργασία αναλαμβάνει να υπολογίσει το αντίστοιχο τμήμα του πίνακα γινομένου. Η διαδικασία, αυτή είναι πολύ πιο αποδοτική σε σχέση με τον παραδοσιακό τρόπο υπολογισμού του γινομένου πινάκων. Ο τρόπος υπολογισμού του γινομένου των δύο πινάκων με την παραπάνω μέθοδο είναι γνωστός ως αλγόριθμος του Fox.

Για λόγους απλότητας θεωρούμε πως οι πίνακες A, B και C είναι τετραγωνικοί πίνακες τάξεως N, ενώ το πλήθος των διεργασιών του συστήματος είναι ίσο με N^2. Δεδομένου ότι το πλήθος των στοιχείων ενός

τετραγωνικού πίνακα NxN είναι και αυτό ίσο με N^2, κάθε μια από τις παραπάνω διεργασίες θα αναλάβει τον υπολογισμό ενός και μοναδικού στοιχείου του πίνακα γινομένου. Θεωρώντας πως οι διεργασίες είναι τοποθετημένες στους κόμβους ενός δυσδιάστατου πλέγματος με διαστάσεις NxN, είναι προφανές πως η διεργασία της καρτεσιανής τοπολογίας με συντεταγμένες (i,j) θα αναλάβει τον υπολογισμό του στοιχείου Cij του πίνακα C (i,j=0,1,2,…,N-1) (σχήμα 33).



Σχήμα 33. *Η διαδικασία υπολογισμού των στοιχείων δυσδιάστατου πίνακα με τον αλγόριθμο του Fox*

Η διαδικασία υπολογισμού του στοιχείου C_{ij} από τη διεργασία (i,j) πραγματοποιείται με τη χρήση ενός αλγορίθμου N βημάτων:

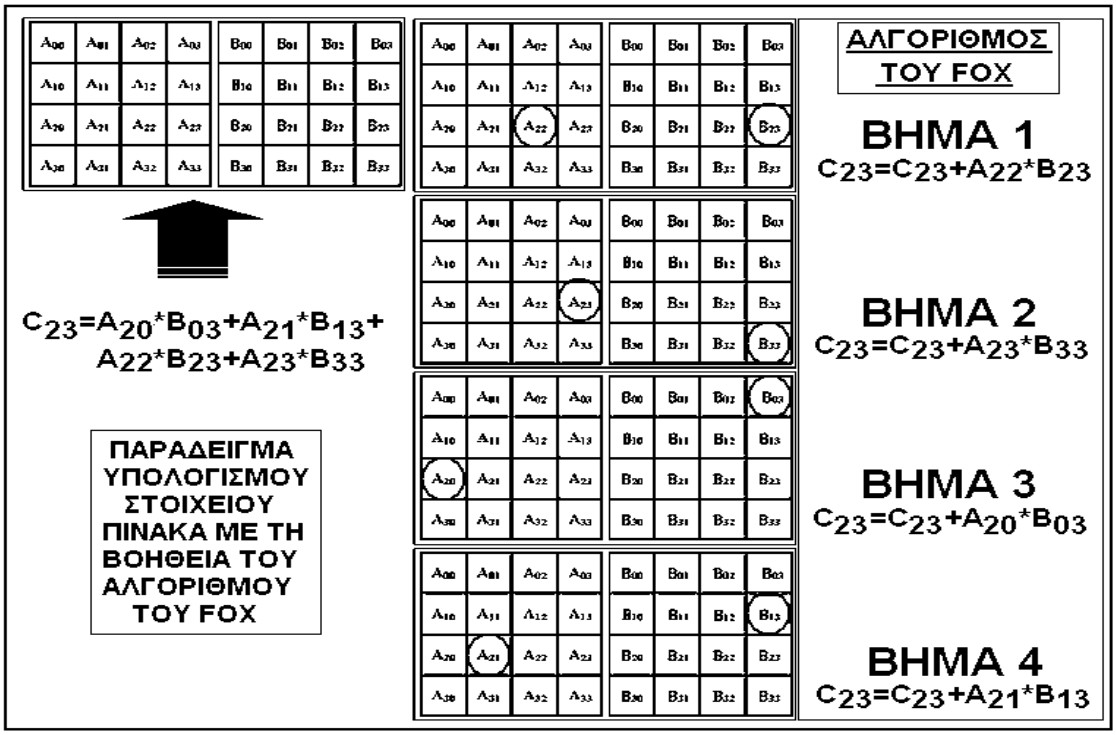
Στο πρώτο βήμα η διεργασία (i,j) ταυτοποιεί το διαγώνιο στοιχείο της υπ' αριθμόν i γραμμής – δηλαδή το A_{ii} – και το πολλαπλασιάζει με το στοιχείο B_{ij} του πίνακα B.

Στο δεύτερο βήμα υπολογίζεται το γινόμενο του αμέσως επόμενου στοιχείου της υπ' αριθμόν i γραμμής (που είναι το στοιχείο $A_{i,i+1}$) με το στοιχείο του πίνακα B που βρίσκεται ακριβώς κάτω από το προηγούμενο (δηλαδή το στοιχείο $B_{i+1,j}$). Μιλώντας γενικά, στο υπ' αριθμόν k βήμα της διαδικασίας, θα λάβει χώρα ο υπολογισμός του γινομένου $A_{i,i+k}$*$B_{i+k,j}$. Αν προσθέσουμε όλα αυτά τα επιμέρους γινόμενα, θα υπολογίσουμε την τιμή του στοιχείου c_{ij} σύμφωνα με την εξίσωση ορισμού του που είναι η

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj} \quad (7.2)$$

Η διαδικασία υπολογισμού των στοιχείων του πίνακα γινομένου χρησιμοποιώντας τον αλγόριθμο του Fox, παρουσιάζεται στο σχήμα 34. Αν θεωρήσουμε ότι το σύστημα περιλαμβάνει 16 διεργασίες οι οποίες βρίσκονται τοποθετημένες σε ένα δυσδιάστατο καρτεσιανό πλέγμα με διαστάσεις 4x4, είναι προφανές πως ο υπολογισμός του στοιχείου C_{ij} θα πραγματοποιηθεί από τη διεργασία του καρτεσιανού πλέγματος με συντεταγμένες (i,j). Στο σχήμα που ακολουθεί (σχήμα 34), παρουσιάζεται η διαδικασία υπολογισμού του στοιχείου C_{23} από τη διεργασία με συντεταγμένες (2,3). Επειδή η τιμή του στοιχείου C_{23} πρόκειται να υπολογισθεί ως ένα άθροισμα τεσσάρων επιμέρους γινομένων, θα πρέπει να λάβει μια αρχική τιμή ίση με το μηδέν. Στο επόμενο βήμα της διαδικασίας, η τιμή αυτού του στοιχείου υπολογίζεται σε τέσσερα βήματα ως το άθροισμα:

$$C_{23} = A_{22}*B_{23} + A_{23}*B_{33} + A_{20}*B_{03} + A_{21}*B_{13}.$$



Σχήμα 34. *Σχηματική αναπαράσταση του αλγορίθμου του Fox*

Για τις διεργασίες εκείνες οι οποίες βρίσκονται στο εσωτερικό του πλέγματος, η διαδικασία υπολογισμού του στοιχείου που αντιστοιχεί σε αυτές περιλαμβάνει την ανακύκλωση των τιμών του ενδιάμεσου δείκτη k, μέχρι που επανέρχεται στην αρχική μηδενική του τιμή. Στο παραπάνω παράδειγμα, οι πίνακες που πολλαπλασιάζονται είναι τετραγωνικοί πίνακες διαστάσεων 4x4, δηλαδή ο δείκτης k παίρνει τις τιμές 0,1,2,3. Εάν σε κάποιο βήμα της διαδικασίας ο δείκτης αυτός λάβει τη μέγιστη τιμή του – δηλαδή την τιμή 3 – και η διαδικασία δεν έχει ακόμη ολοκληρωθεί, τότε στο επόμενο στάδιο, ο δείκτης θα επανέλθει στην τιμή 0, παραπέμποντας έτσι σε στοιχεία των πινάκων που βρίσκονται στην ίδια γραμμή ή στήλη αλλά προς την αντίθετη κατεύθυνση. Μιλώντας γενικά, εάν η διεργασία με

συντεταγμένες (i,j) εντοπίζεται στο εσωτερικό του καρτεσιανού πλέγματος, η τιμή του στοιχείου C_{ij} θα προκύψει ως το άθροισμα:

$$Cij=Aii*Bij+A_{i,i+1}*B_{i+1,j}+…+A_{i,N-1}*B_{N-1,j}+A_{i0}*B_{0j}+…+Ai,i\text{-}1B_{i-1,j}$$

Για να αντιμετωπίσουμε τις παραπάνω ανακυκλώσεις των τιμών του ενδιάμεσου δείκτη, καθώς επίσης και για να απομακρύνουμε κάθε ενδεχόμενο αναφοράς μας σε περιοχές δεικτών που δεν είναι έγκυρες, μπορούμε να εφαρμόσουμε την εξής πρακτική: στη διαδικασία υπολογισμού των μερικών γινομένων δεν χρησιμοποιούμε την τιμή (i+k) – για το υπ' αριθμόν k βήμα της διαδικασίας – αλλά το υπόλοιπο της διαίρεσης αυτής της ποσότητας με την τάξη N των τετραγωνικών πινάκων. Εάν λοιπόν υπολογίσουμε την ποσότητα m=(i+k) MOD N και στη συνέχεια υπολογίσουμε την τιμή του στοιχείου C_{ij} – για το υπ' αριθμόν k βήμα – από τη σχέση $C_{ij}=C_{ij}+A_{im}*B_{mk}$ – τότε διασφαλίζουμε την ομαλή λειτουργία του αλγορίθμου χωρίς την εμφάνιση των προβλημάτων που παρουσιάστηκαν παραπάνω. Η αναλυτική διαδικασία υπολογισμού του στοιχείου C_{ij} για ένα τετραγωνικό πίνακα με διαστάσεις NxN σε μαθηματική διατύπωση έχει τη μορφή που ακολουθεί στη συνέχεια.

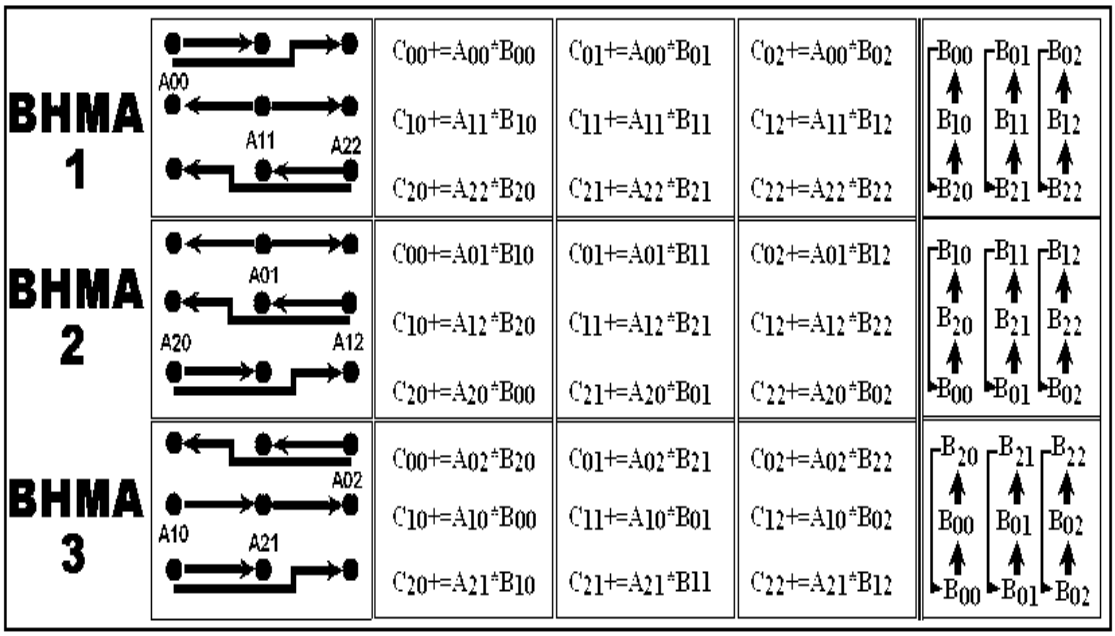
Βήμα 0:	$C_{ij}=A_{ii}*B_{ij}$	
Βήμα 1:	$C_{ij}=C_{ij}+A_{im}*B_{mj}$	m=(i+1) mod N
Βήμα 2:	$C_{ij}=C_{ij}+A_{im}*B_{mj}$	m=(i+2) mod N
…….		
Βήμα k:	$C_{ij}=C_{ij}+A_{im}*B_{mj}$	m=(i+k) mod N
…….		
Βήμα N-1:	$C_{ij}=C_{ij}+A_{im}*B_{mj}$	m=(i+N-1) mod N

Πρέπει να σημειωθεί πως κατά τη διάρκεια του υπ' αριθμόν k βήματος υπολογισμού του στοιχείου C_{ij}, απαιτείται η γνώση του στοιχείου A_{im} (m=(i+k) mod N) από όλες τις διεργασίες που υπολογίζουν στοιχεία του πίνακα γινομένου που ανήκουν στην ίδια γραμμή i. Λαμβάνοντας υπ' όψιν πως κατά την εκκίνηση της διαδικασίας χρησιμοποιείται για τον υπολογισμό του πρώτου μερικού γινομένου η τιμή του στοιχείου A_{ii}, είναι προφανές πως αυτό το στοιχείο θα πρέπει να διαβιβαστεί σε όλες αυτές τις διεργασίες, πριν την πραγματοποίηση των παραπάνω πολλαπλασιασμών, διαδικασία που λαμβάνει χώρα δια της χρήσεως της συνάρτησης εκπομπής, **MPI_Bcast**. Ομοίως, κατά την πρώτη φάση της διαδικασίας υπολογισμού του στοιχείου C_{ij}, χρησιμοποιείται το αντίστοιχο στοιχείο B_{ij} του πίνακα B, το οποίο, μετά την ολοκλήρωση του βήματος θα πρέπει να διαβιβαστεί στη διεργασία που βρίσκεται στη θέση (i-1,j), δηλαδή ακριβώς πάνω από την τρέχουσα διεργασία. Ταυτόχρονα, η διεργασία (i,j) θα πρέπει να παραλάβει το στοιχείο $B_{i+1,j}$ προκειμένου να υπολογίσει την τιμή του επόμενου μερικού γινομένου.

Παρατηρούμε δηλαδή πως ο υπολογισμός του γινομένου δύο πινάκων δια της χρήσεως του αλγορίθμου του Fox, περιλαμβάνει:

- μια διαδικασία εκπομπής του στοιχείου A_{ii} σε όλες τις διεργασίες της γραμμής υπ' αριθμόν i, και
- μια διαδικασία μετάδοσης του στοιχείου B_{ij} κατά την κατακόρυφη διεύθυνση, δηλαδή κατά μήκος της στήλης j και από κάτω προς τα πάνω.

Στο παρακάτω σχήμα (σχήμα 35) παρουσιάζονται οι διαδικασίες διακίνηση δεδομένων κατά την εκτέλεση του αλγορίθμου του Fox για τετραγωνικούς πίνακες διαστάσεων 3x3. Ο πολλαπλασιασμός των πινάκων στην περίπτωση αυτή πραγματοποιείται από ένα δυσδιάστατο καρτεσιανό πλέγμα 9 διεργασιών.



Σχήμα 35. *Διαδικασίες διακίνησης δεδομένων στον αλγόριθμο του Fox*

Είναι προφανές πως η υλοποίηση του αλγορίθμου του Fox με τον παραπάνω τρόπο δεν είναι αποδοτική για πίνακες μεγάλης τάξης, καθώς για τετραγωνικούς πίνακες τάξεως N απαιτούνται συνολικά $p=N^2$ διεργασίες. Για να αντιμετωπίσουμε το πρόβλημα αυτό μπορούμε αντί να αντιστοιχούμε σε κάθε διεργασία ένα και μοναδικό στοιχείο του πίνακα, να συσχετίζουμε με αυτή ένα σύνολο στοιχείων του αρχικού πίνακα, δηλαδή ένα υπο-πίνακα. Στο επόμενο βήμα της διαδικασίας, εφαρμόζουμε τη μέθοδο που παρουσιάσαμε στις προηγούμενες σελίδες χρησιμοποιώντας αυτούς τους υπο-πίνακες. Επίσης προκειμένου να διασφαλίσουμε πως οι διεργασίες που θα χρησιμοποιήσουμε θα σχηματίζουν μια τετραγωνική καρτεσιανή τοπολογία, μπορούμε να επιλέξουμε το πλήθος των διεργασιών, p, με τέτοιο τρόπο, ώστε η τετραγωνική του ρίζα, $\sqrt{p}$ να διαιρεί ακριβώς την τάξη των πινάκων, N. Σε αυτή την περίπτωση, η κάθε διεργασία συσχετίζεται με τρεις τετραγωνικούς υπο-πίνακες – για τους αρχικούς πίνακες A,B και C – με διαστάσεις KxK όπου $K=N/\sqrt{p}$. Έτσι για N=p=4 η διάσταση αυτών των υπο-πινάκων θα είναι K=2 και αυτοί οι υπο-πίνακες θα έχουν τη μορφή:

$$A'_{00} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \qquad A'_{01} = \begin{pmatrix} A_{02} & A_{03} \\ A_{12} & A_{13} \end{pmatrix}$$

$$A'_{10} = \begin{pmatrix} A_{20} & A_{21} \\ A_{30} & A_{31} \end{pmatrix} \qquad A'_{11} = \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix}$$

Εάν λοιπόν ορίσουμε με παρόμοιο τρόπο τους υπο-πίνακες B'_{ij} και $C'_{ij,}$ τους συσχετίσουμε με τη διεργασία (i,j) και ορίσουμε την ποσότητα $q=\sqrt{p}$, τότε, χρησιμοποιώντας το αλγόριθμο του Fox θα υπολογίσουμε τα στοιχεία του υπο-πίνακα C'_{ij} από τη σχέση

$$C'_{ij} = A'_{ii}*B'_{ij} + A'_{i,i+1}*B'_{i+1,j} +\ldots+ A'_{i,q-1}*B'_{q-1,\,j}+ A'_{i0}*B'_{0j} +\ldots+ A'_{i,i-1}*B'_{i-1,j}$$

Όπως μπορούμε πολύ εύκολα να διαπιστώσουμε, το αποτέλεσμα που λαμβάνουμε σε αυτή την περίπτωση, είναι το ίδιο με εκείνο που παίρνουμε χρησιμοποιώντας μία διεργασία για κάθε στοιχείο των πινάκων Α, Β και C.

7.1. Υλοποίηση του αλγορίθμου Fox με χρήση του MPI

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define ORDER 12

typedef struct GridInfo
{
    int processNumber;      // total number of processes
    MPI_Comm gridComm;      // communicator for entire grid
    MPI_Comm rowComm;       // communicator for current row
    MPI_Comm colComm;       // communicator for current column
    int gridOrder;          // the order of grid
    int currentRow;         // row of current process
    int currentCol;         // column of current process
    int gridCommRank;       // rank of current process in gridComm
} GridInfo;

void PrintGridInfo(GridInfo * grid)
{
    printf("Number of Processes is %d\n", grid->processNumber);
    printf("Grid Comm Identifier is %d\n", grid->gridComm);
    printf("Row Comm Identifier is %d\n", grid->rowComm);
    printf("Column Comm Identifier is %d\n", grid->colComm);
    printf("Grid Order is %d\n", grid->gridOrder);
    printf("Current Process Coordinates are(%d, %d)\n",
    grid->currentRow, grid->currentCol);
    printf("Process rank in Grid is %d\n", grid->gridCommRank);
}

double** Allocate2DMatrix(int rows, int columns)
{
    int counter;
    double** matrix;
    matrix =(double**)malloc(rows*sizeof(double *));
    if(!matrix)
        return(NULL);
    for(counter=0; counter<rows; counter++)
    {
        matrix[counter] =(double*)malloc(columns*sizeof(double));
        if(!matrix[counter])
            return(NULL);
    }
    return matrix;
}

void Free2DMatrix(double** matrix, int rows)
{
    int counter;
    for(counter=0; counter<rows; counter++)
        free((double*)matrix[counter]);
        free((double**)matrix);
}

int SetupGrid(GridInfo * grid)
{
    int flag, worldRank;
    int dims[2], periods[2], gridCoords[2], subCoords[2];

    MPI_Initialized(&flag);
    if(flag == 0)
        return(-1);

    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->processNumber));

    if(sqrt((double)grid->processNumber) == (int)sqrt((double)grid->processNumber))
        grid->gridOrder = (int)sqrt((double)grid->processNumber);
    else
    {
        MPI_Finalize();
        return(-2);
    }

    dims[0]=dims[1]=grid->gridOrder;
    periods[0]=periods[1]=1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &(grid->gridComm));

    MPI_Comm_rank(grid->gridComm, &(grid->gridCommRank));
    MPI_Cart_coords(grid->gridComm, grid->gridCommRank, 2, gridCoords);
    grid->currentRow=gridCoords[0];
    grid->currentCol=gridCoords[1];

    subCoords[0]=0;
    subCoords[1]=1;
    MPI_Cart_sub(grid->gridComm, subCoords, &(grid->rowComm));

    subCoords[0]=1;
    subCoords[1]=0;
    MPI_Cart_sub(grid->gridComm, subCoords, &(grid->colComm));

    return(0);
}
```

```
void LocalMatrixProduct(double ** A, double ** B, double ** C, int dim)
{
    int i,j,k;
    for(i=0;i<dim;i++)
    {
        for(j=0;j<dim;j++)
        {
            for(k=0;k<dim;k++)
            {
                C[i][j]+=A[i][k]*B[k][j];
            }
        }
    }
}

int Scatter2DMatrix(double ** matrix, int N, double ** local, int dim, int root, GridInfo * grid)
{
    int flag, rank, loops = N/dim;
    int size, dest, packPosition;
    int counter, index, coords[2], i, j;
    double * tempArray;
    MPI_Status status;

    MPI_Initialized(&flag);
    if(flag == 0)
        return(-1);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if((root<0)||(root>=size))
    {
        MPI_Finalize();
        return(-1);
    }

    tempArray =(double *) malloc(dim*dim*sizeof(double));
    if(!tempArray)
        return(-1);

    if(rank==root)
    {
        for(counter=0;counter<loops;counter++)
        {
            coords[0]=counter;
            for(index=0;index<loops;index++)
            {
                coords[1]=index;
                MPI_Cart_rank(grid->gridComm, coords, &dest);
                packPosition = 0;
                for(i=dim*counter;i<dim*(counter+1);i++)
                    for(j=dim*index;j<dim*(index+1);j++)
                        MPI_Pack(&matrix[i][j], 1, MPI_DOUBLE, tempArray, 256, &packPosition, MPI_COMM_WORLD);
                if(dest!=root)
                    MPI_Send(tempArray, dim*dim, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
                else
                {
                    for(i=0;i<dim;i++)
                        for(j=0;j<dim;j++)
                            local[i][j]=tempArray[i*dim+j];
                }
            }
        }
    }
    else
    {
        MPI_Recv(tempArray, dim*dim, MPI_DOUBLE, root, 0, MPI_COMM_WORLD, &status);
        for(counter=0;counter<dim;counter++)
            for(index=0;index<dim;index++)
                local[counter][index]=tempArray[counter*dim+index];
    }

    free(tempArray);
    return(0);
}
```

```
int Gather2DMatrix(double ** matrix, int N, double ** local, int dim, int root, GridInfo * grid)
{
    int flag, rank, loops = N/dim;
    int size, cnt, source, rootCoords[2];
    int counter, index, coords[2], i, j;
    double * tempArray;
    MPI_Status status;

    MPI_Initialized(&flag);
    if(flag == 0)
        return(-1);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if((root<0)||(root>=size))
    {
        MPI_Finalize();
        return(-1);
    }
    tempArray =(double *) malloc(dim*dim*sizeof(double));
    if(!tempArray)
        return(-1);

    if(rank==root)
    {
        for(counter=0;counter<loops;counter++)
        {
            coords[0]=counter;
            for(index=0;index<loops;index++)
            {
                coords[1]=index;
                MPI_Cart_rank(grid->gridComm, coords, &source);
                if(source==root)
                {
                    MPI_Cart_coords(grid->gridComm, rank, 2, rootCoords);
                    for(i=dim*rootCoords[0];i<dim*(rootCoords[0]+1);i++)
                    {
                        for(j=dim*rootCoords[1];j<dim*(rootCoords[1]+1);j++)
                        {
                            matrix[i][j]=local[i][j];
                        }
                    }
                }
                else
                {
                    MPI_Recv(tempArray, dim*dim, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, &status);
                    cnt=0;
                    for(i=dim*counter;i<dim*(counter+1);i++)
                    {
                        for(j=dim*index;j<dim*(index+1);j++)
                        {
                            matrix[i][j]=tempArray[cnt];
                            cnt++;
                        }
                    }
                }
            }
        }
    }
    else
    {
        cnt=0;
        for(i=0;i<dim;i++)
            for(j=0;j<dim;j++)
            {
                tempArray[cnt]=local[i][j];
                cnt++;
            }
            MPI_Send(tempArray, dim*dim, MPI_DOUBLE, root, 0, MPI_COMM_WORLD);
    }

    free(tempArray);
    return(0);
}

int Fox(int matrixOrder, GridInfo * grid, double ** localA, double ** localB, double ** localC)
{
    int i,sourceRank, x,y, destRank, root, cnt;
    int dim=matrixOrder/grid->gridOrder;
    double ** tempMatrix;
    MPI_Status status;
    double * sendVec, * recvVec;

    MPI_Cart_shift(grid->colComm, 0, 1, &destRank, &sourceRank);

    tempMatrix=Allocate2DMatrix(dim,dim);
    sendVec =(double *) malloc(dim*dim*sizeof(double));
    recvVec =(double *) malloc(dim*dim*sizeof(double));

    if((!tempMatrix)||(!sendVec)||(!recvVec))
    {
        MPI_Finalize();
        return(-1);
    }
```

```
    for(x=0;x<dim;x++)
        for(y=0;y<dim;y++)
            localC[x][y]=0.0;

    for(i=0;i<grid->gridOrder;i++)
    {
        root=(grid->currentRow+i)%grid->gridOrder;
        if(root==grid->currentCol)
        {
            cnt=0;
            for(x=0;x<dim;x++)
            {
                for(y=0;y<dim;y++)
                {
                    sendVec[cnt]=localA[x][y];
                    cnt++;
                }
            }
            MPI_Bcast(sendVec, dim*dim, MPI_DOUBLE, root, grid->rowComm);
            LocalMatrixProduct(localA, localB, localC, dim);
        }
        else
        {
            MPI_Bcast(recvVec, dim*dim, MPI_DOUBLE, root, grid->rowComm);
            for(x=0;x<dim;x++)
                for(y=0;y<dim;y++)
                    tempMatrix[x][y]=recvVec[x*dim+y];
            LocalMatrixProduct(tempMatrix, localB, localC, dim);
        }

        cnt=0;
        for(x=0;x<dim;x++)
            for(y=0;y<dim;y++)
            {
                sendVec[cnt]=localB[x][y];
                cnt++;
            }

        MPI_Sendrecv_replace(sendVec, dim*dim, MPI_DOUBLE, destRank, 0, sourceRank, 0, grid->colComm, &status);

        for(x=0;x<dim;x++)
            for(y=0;y<dim;y++)
                localB[x][y]=sendVec[x*dim+y];
    }

    Free2DMatrix(tempMatrix, dim);
    free(sendVec);
    free(recvVec);
    return(0);
}

int main(int argc, char ** argv)
{
    int i,j,rank, dim, root=0;
    GridInfo grid;
    double ** localA, ** localB, ** localC;
    double ** aMatrix, ** bMatrix, ** cMatrix;

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

    SetupGrid(&grid);
    dim = ORDER/grid.gridOrder;

    localA=Allocate2DMatrix(dim,dim);
    if(localA==NULL)
        return(-1);
    localB=Allocate2DMatrix(dim,dim);
    if(localB==NULL)
        return(-1);
    localC=Allocate2DMatrix(dim,dim);
    if(localC==NULL)
        return(-1);

    if(rank==root)
    {
        aMatrix=Allocate2DMatrix(ORDER,ORDER);
        bMatrix=Allocate2DMatrix(ORDER,ORDER);
        cMatrix=Allocate2DMatrix(ORDER,ORDER);

        if((!aMatrix)||(!bMatrix)||(!cMatrix))
        {
            printf("Not enough memory for matric allocation. Aborting...\n");
            MPI_Finalize();
            return(-1);
        }

        for(i=0;i<ORDER;i++)
        {
            for(j=0;j<ORDER;j++)
            {
                aMatrix[i][j]=(double)rand()/RAND_MAX;
                bMatrix[i][j]=(double)rand()/RAND_MAX;
                cMatrix[i][j]=0.0;
            }
        }
```

```
        printf("\n\nMatrix A data:\n");
        for(i=0;i<ORDER;i++)
        {
            for(j=0;j<ORDER;j++)
                printf("%.3f ", aMatrix[i][j]);
            printf("\n");
        }

        printf("\n\nMatrix B data:\n");
        for(i=0;i<ORDER;i++)
        {
            for(j=0;j<ORDER;j++)
                printf("%.3f ", bMatrix[i][j]);
            printf("\n");
        }
    }

    Scatter2DMatrix(aMatrix, ORDER, localA, dim, root, &grid);
    Scatter2DMatrix(bMatrix, ORDER, localB, dim, root, &grid);

    Fox(ORDER, &grid, localA, localB, localC);

    Gather2DMatrix(cMatrix, ORDER, localC, dim, root, &grid);

    if(rank==root)
    {
        printf("\n\nMatrix C data:\n");
        for(i=0;i<ORDER;i++)
        {
            for(j=0;j<ORDER;j++)
                printf("%.3f ", cMatrix[i][j]);
            printf("\n");
        }
    }

    Free2DMatrix(localA, dim);
    Free2DMatrix(localB, dim);
    Free2DMatrix(localC, dim);

    MPI_Barrier(MPI_COMM_WORLD);

    if(rank==root)
    {
        Free2DMatrix(aMatrix, ORDER);
        Free2DMatrix(bMatrix, ORDER);
        Free2DMatrix(cMatrix, ORDER);
    }

    MPI_Comm_free(&grid.gridComm);
    MPI_Comm_free(&grid.rowComm);
    MPI_Comm_free(&grid.colComm);
    MPI_Finalize();
    return(0);
}
```

7.1.1. Επεξήγηση του πηγαίου κώδικα

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define ORDER 12
```

Η σταθερά **ORDER** αναφέρεται στην τάξη των τετραγωνικών πινάκων A και B η οποία στο παράδειγμά μας έχει την τιμή 12.

```
typedef struct GridInfo
{
        int processNumber;          // total number of processes
        MPI_Comm gridComm;          // communicator for entire grid
        MPI_Comm rowComm;           // communicator for current row
        MPI_Comm colComm;           // communicator for current column
        int gridOrder;              // the order of grid
        int currentRow;             // row of current process
        int currentCol;             // column of current process
        int gridCommRank;           // rank of current process in gridComm
} GridInfo;
```

Η δομή δεδομένων **GridInfo** χρησιμοποιείται για την περιγραφή του καρτεσιανού πλέγματος και για την κάθε διεργασία. Η ακέραια μεταβλητή **processNumber** χρησιμοποιείται για την αποθήκευση του αριθμού των διεργασιών. Στις μεταβλητές **gridComm**, **rowComm** και **colComm** αποθηκεύεται ο communicator για όλο το πλέγμα, για την τρέχουσα γραμμή και για την τρέχουσα στήλη αντίστοιχα. Η μεταβλητή **gridOrder** χρησιμοποιείται για την αποθήκευση της τάξης του πλέγματος**.** Στις μεταβλητές **currentRow** και **currentCol** αποθηκεύεται ο αριθμός της γραμμής και στήλης αντίστοιχα που αντιστοιχεί στην τρέχουσα διεργασία. Τέλος, στη μεταβλητή **gridCommRank** αποθηκεύεται η τάξη της τρέχουσας διεργασίας στον communicator **gridComm**.

Η αρχικοποίηση της παραπάνω δομής θα πραγματοποιηθεί από την κάθε διεργασία ξεχωριστά. Οι μεταβλητές **processNumber**, **gridOrder** και **gridComm** θα έχουν την ίδια τιμή για όλες τις διεργασίες, ενώ η κάθε διεργασία θα έχει τη δική της τάξη και τις δικές της συντεταγμένες στη δυσδιάστατη καρτεσιανή τοπολογία.

```
void PrintGridInfo(GridInfo * grid)
{
        printf("Number of Processes is %d\n", grid->processNumber);
        printf("Grid Comm Identifier is %d\n", grid->gridComm);
        printf("Row Comm Identifier is %d\n", grid->rowComm);
        printf("Column Comm Identifier is %d\n", grid->colComm);
        printf("Grid Order is %d\n", grid->gridOrder);
        printf("Current Process Coordinates are(%d, %d)\n",
        grid->currentRow, grid->currentCol);
        printf("Process rank in Grid is %d\n", grid->gridCommRank);
}
```

Η συνάρτηση **PrintGridInfo** παίρνει ως όρισμα ένα αντικείμενο της δομής **GridInfo** και τυπώνει στην οθόνη τις τιμές των μεταβλητών του συγκεκριμένου αντικειμένου.

```
double** Allocate2DMatrix(int rows, int columns)
{
	int counter;
	double** matrix;
	matrix =(double**)malloc(rows*sizeof(double *));
	if(!matrix)
		return(NULL);
	for(counter=0; counter<rows; counter++)
	{
		matrix[counter] =(double*)malloc(columns*sizeof(double));
		if(!matrix[counter])
			return(NULL);
	}
	return matrix;
}
```

Η παραπάνω συνάρτηση **Allocate2DMatrix** δημιουργεί και επιστρέφει έναν πίνακα δύο διαστάσεων για αποθήκευση αριθμών κινητής υποδιαστολής. Ο αριθμός των γραμμών και των στηλών δίνεται ως όρισμα στη συνάρτηση και μέσω της συνάρτησης **malloc** δεσμεύεται ο απαραίτητος χώρος μνήμης για το νέο πίνακα.

```
void Free2DMatrix(double** matrix, int rows)
{
	int counter;
	for(counter=0; counter<rows; counter++)
		free((double*)matrix[counter]);
		free((double**)matrix);
}
```

Η συνάρτηση **Free2DMatrix** δέχεται ως όρισμα έναν πίνακα δύο διαστάσεων και τον πλήθος των γραμμών του και αποδεσμεύει τη μνήμη που έχει δεσμευτεί για την αποθήκευση των δεδομένων του συγκεκριμένου πίνακα. Η αποδέσμευση της μνήμης γίνεται με χρήση της συνάρτησης **free**.

Η αρχικοποίηση της δομής **GridInfo** για κάθε μια από τις διεργασίες της εφαρμογής πραγματοποιείται μέσω της συνάρτησης **SetupGrid,** η υλοποίηση της οποίας παρουσιάζεται στον κώδικα που ακολουθεί:

```
int SetupGrid(GridInfo * grid)
{
	int flag, worldRank;
```

Η ακέραια μεταβλητή **flag** χρησιμοποιείται για να αποθηκεύσουμε το αποτέλεσμα της συνάρτησης **MPI_Initialized.**

```
	int dims[2], periods[2], gridCoords[2], subCoords[2];

	MPI_Initialized(&flag);
```

```
if(flag == 0)
        return(-1);
```

Μέσω του παραπάνω ελέγχου, αν δεν έχει προηγουμένως λάβει χώρα η αρχικοποίηση του περιβάλλοντος MPI (η οποία γίνεται χρησιμοποιώντας τη συνάρτηση **MPI_Init**), η συνάρτηση επιστρέφει -1 δηλώνοντας έτσι ότι έχει συμβεί κάποιο σφάλμα.

```
MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
MPI_Comm_size(MPI_COMM_WORLD, &(grid->processNumber));
```

Η τάξη της τρέχουσας διεργασίας και το συνολικό πλήθος των διεργασιών του προεπιλεγμένου Communicator που περιγράφεται από τη σταθερά **MPI_COMM_WORLD** αποθηκεύονται στις μεταβλητές **worldRank** και **processNumber** (του αντικειμένου **grid**) τύπου integer καλώντας τις παραπάνω συναρτήσεις αντίστοιχα.

```
if(sqrt((double)grid->processNumber) == (int)sqrt((double)grid-
>processNumber))
        grid->gridOrder = (int)sqrt((double)grid->processNumber);
else
{
        MPI_Finalize();
        return(-2);
}
```

Στην τάξη του πλέγματος που περιγράφεται από τη μεταβλητή **gridOrder** δίνεται τιμή ίση με την τετραγωνική ρίζα του αριθμού των διεργασιών. Σε περίπτωση που η τιμή του πεδίου **processNumber** δεν είναι τέλειο τετράγωνο, η συνάρτηση διακόπτει την εκτέλεσή της.

Στο επόμενο βήμα λαμβάνει χώρα η δημιουργία της καρτεσιανής τοπολογίας και του communicator **gridComm** που συσχετίζεται με αυτή. Η δημιουργία αυτή στηρίζεται στη χρήση της συνάρτησης **MPI_Cart_create**, η οποία καλείται με ορίσματα int **dims**[2]={**gridOrder**, **gridOrder**} για τις διαστάσεις του καρτεσιανού πλέγματος, int **periods**[2]={true, true} (η τοπολογία πρέπει υποχρεωτικά να είναι περιοδική τουλάχιστον ως προς την κατακόρυφη διεύθυνση έτσι ώστε να είναι δυνατή η κυκλική μετάδοση των στοιχείων B_{mj}), και **reorder**=true (έτσι ώστε να λάβει χώρα επαναδιάταξη των διεργασιών στον communicator **gridComm** όσον αφορά τις τιμές των τάξεών τους).

```
dims[0]=dims[1]=grid->gridOrder;
periods[0]=periods[1]=1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &(grid-
>gridComm));
```

Μετά τη δημιουργία του δυσδιάστατου καρτεσιανού πλέγματος, η κάθε διεργασία ανακτά τη νέα τιμή της τάξης της στον communicator **gridComm** καθώς και τις συντεταγμένες (i,j) που ταυτοποιούν τη θέση της πάνω στο καρτεσιανό πλέγμα, και καταχωρεί αυτές τις τιμές στα πεδία **gridCommRank**, **currentRow** και **currentColumn** αντίστοιχα της δομής **GridInfo**. Η ανάκτηση αυτών των πληροφοριών γίνεται κατά τα γνωστά

δια της χρήσεως των συναρτήσεων **MPI_Comm_rank** και **MPI_Cart_coords**.

```
MPI_Comm_rank(grid->gridComm, &(grid->gridCommRank));
MPI_Cart_coords(grid->gridComm, grid->gridCommRank, 2, gridCoords);
grid->currentRow=gridCoords[0];
grid->currentCol=gridCoords[1];
```

Στο τελευταίο και πιο ενδιαφέρον βήμα της διαδικασίας, η συνάρτηση **SetupGrid** δημιουργεί τους communicators **rowComm** και **colComm** που επιτρέπουν τη διακίνηση της πληροφορίας κατά την οριζόντια και την κατακόρυφη διεύθυνση αντίστοιχα. Η δημιουργία αυτών των communicators γίνεται δια της χρήσεως της συνάρτησης **MPI_Cart_sub** η οποία καλείται δύο φορές. Το όρισμα **subCoords** κατά την πρώτη κλήση της έχει τις τιμές {0,1} με αποτέλεσμα τη δημιουργία τεσσάρων communicators **rowComm** που περιλαμβάνουν τις διεργασίες των αντίστοιχων γραμμών της καρτεσιανής τοπολογίας, ενώ τη δεύτερη φορά το όρισμα **subCoords** έχει τις τιμές {1,0} με αποτέλεσμα τη δημιουργία τεσσάρων communicators **colComm** που αντιστοιχούν στις τέσσερις στήλες αυτής της τοπολογίας – υπενθυμίζουμε πως το δυσδιάστατο καρτεσιανό πλέγμα των διεργασιών έχει διαστάσεις 4x4 και επομένως χαρακτηρίζεται από την ύπαρξη τεσσάρων γραμμών και τεσσάρων στηλών. Είναι προφανές πως όλες οι διεργασίες που ανήκουν στην ίδια γραμμή έχουν την ίδια τιμή στο πεδίο **rowComm** έτσι ώστε να είναι δυνατή η ανταλλαγή της πληροφορίας ανάμεσά τους. Με τον ίδιο τρόπο όλες οι διεργασίες που ανήκουν στην ίδια στήλη έχουν την ίδια τιμή στο πεδίο **colComm** της δομής **GridInfo**.

```
        subCoords[0]=0;
        subCoords[1]=1;
        MPI_Cart_sub(grid->gridComm, subCoords, &(grid->rowComm));

        subCoords[0]=1;
        subCoords[1]=0;
        MPI_Cart_sub(grid->gridComm, subCoords, &(grid->colComm));

        return(0);
}
```

Η συνάρτηση **LocalMatrixProduct** υπολογίζει το γινόμενο των υπο-πινάκων σε κάθε στάδιο του αλγορίθμου.

```
void LocalMatrixProduct(double ** A, double ** B, double ** C, int dim)
{
        int i,j,k;
        for(i=0;i<dim;i++)
        {
                for(j=0;j<dim;j++)
                {
                        for(k=0;k<dim;k++)
                        {
                                C[i][j]+=A[i][k]*B[k][j];
                        }
                }
```

```
        }
}
```

Η συνάρτηση **Scatter2DMatrix** είναι συλλογική, καλείται δηλαδή από όλες τις διεργασίες της εφαρμογής. Ακολουθώντας την ονοματολογία που χρησιμοποιείται για τις συλλογικές συναρτήσεις του MPI, η τάξη της διεργασίας–ρίζα που θα αναλάβει τη διασπορά των δεδομένων του αρχικού πίνακα περιγράφεται από το όρισμα **root** που εμφανίζεται στο πρωτότυπο της συνάρτησης. Όσον αφορά τα υπόλοιπα ορίσματα, τα **N** και **matrix** αναφέρονται στην τάξη και στο όνομα του τετραγωνικού πίνακα που διαμοιράζεται στις διεργασίες της εφαρμογής, τα **dim** και **local** αναφέρονται στην τάξη και στο όνομα του τοπικού υπο-πίνακα της κάθε διεργασίας που θα παραλάβει τα δεδομένα που έχουν διαμοιραστεί, ενώ το αντικείμενο **grid** της δομής **GridInfo** περιέχει τις παραμέτρους της καρτεσιανής τοπολογίας για κάθε μια από τις διεργασίες της εφαρμογής.

```
int Scatter2DMatrix(double ** matrix, int N, double ** local, int dim, int root, GridInfo * grid)
{
        int flag, rank, loops = N/dim;
        int size, dest, packPosition;
        int counter, index, coords[2], i, j;
        double * tempArray;
        MPI_Status status;

        MPI_Initialized(&flag);
        if(flag == 0)
                return(-1);

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

Η τάξη της τρέχουσας διεργασίας και το συνολικό πλήθος των διεργασιών του Communicator που περιγράφεται από τη σταθερά **MPI_COMM_WORLD** αποθηκεύονται στις μεταβλητές **rank** και **size** τύπου integer καλώντας τις παραπάνω συναρτήσεις αντίστοιχα.

```
        if((root<0)||(root>=size))
        {
                MPI_Finalize();
                return(-1);
        }
```

Σε περίπτωση που η τάξη της διεργασίας–ρίζα (περιγράφεται από το όρισμα **root**) που θα αναλάβει τη διασπορά των δεδομένων του αρχικού πίνακα δεν έχει έγκυρη τιμή (*if((root<0)||(root>=size)))*, το πρόγραμμα τερματίζει τη λειτουργία του.

```
        tempArray =(double *) malloc(dim*dim*sizeof(double));
```

Μέσω της συνάρτησης **malloc**, δεσμεύεται για το διάνυσμα **tempArray** ο απαραίτητος χώρος στη μνήμη για την αποθήκευση dim*dim αριθμών κινητής υποδιαστολής.

```
if(!tempArray)
        return(-1);

if(rank==root)
{
        for(counter=0;counter<loops;counter++)
        {
                coords[0]=counter;
                for(index=0;index<loops;index++)
                {
                        coords[1]=index;
                        MPI_Cart_rank(grid->gridComm, coords, &dest);
                        packPosition = 0;
                        for(i=dim*counter;i<dim*(counter+1);i++)
                                for(j=dim*index;j<dim*(index+1);j++)
                                        MPI_Pack(&matrix[i][j], 1,
                                        MPI_DOUBLE, tempArray, 256,
                                        &packPosition, MPI_COMM_WORLD);
```

Η διασπορά των στοιχείων των πινάκων Α και Β στηρίζεται στη χρήση της συνάρτησης **MPI_Pack**, γεγονός που είναι αναμενόμενο, καθώς τα στοιχεία των υπο-πινάκων που θα αποσταλούν καταλαμβάνουν μη συνεχείς θέσεις μνήμης – εναλλακτικά θα μπορούσε να χρησιμοποιηθεί για τον ίδιο σκοπό η συνάρτηση **MPI_Type_vector** η οποία θα όριζε και τον κατάλληλο τύπο δεδομένων. Η συνάρτηση **MPI_Pack** στον παραπάνω βρόγχο συσκευάζει τα σωστά στοιχεία του αρχικού πίνακα σε μια συνεχή περιοχή μνήμης και τα αποστέλλει προς τις διεργασίες του συστήματος οι οποίες και τα παραλαμβάνουν καλώντας τη συνάρτηση **MPI_Recv**. Αυτή η διαδικασία αποστολής δεδομένων πραγματοποιείται από τη διεργασία–ρίζα, η οποία προφανώς δεν αποστέλλει τα δικά της στοιχεία στον εαυτό της αλλά τα μεταφέρει απευθείας στο δικό της αντίγραφο των πινάκων **localA** και **localB**.

```
                        if(dest!=root)
                                MPI_Send(tempArray, dim*dim,
                                MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
                        else
                        {
                                for(i=0;i<dim;i++)
                                        for(j=0;j<dim;j++)
                                                local[i][j]=tempArray[i*dim+j];
                        }
                }
        }
}
else
{
        MPI_Recv(tempArray, dim*dim, MPI_DOUBLE, root, 0,
        MPI_COMM_WORLD, &status);
        for(counter=0;counter<dim;counter++)
                for(index=0;index<dim;index++)

                        local[counter][index]=tempArray[counter*dim+index
                        ];
```

Μέσω της συνάρτησης **MPI_Recv** όλες οι διεργασίες (εκτός από διεργασία-ρίζα) παραλαμβάνουν τα στοιχεία του αρχικού πίνακα και τα μεταφέρουν στο δικό τους αντίγραφο των πινάκων **localA** και **localB**.

```
        }

        free(tempArray);
```

Η συνάρτηση **free** απελευθερώνει τον χώρο που είχε δεσμευτεί στη μνήμη για το διάνυσμα **tempArray**.

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

Στο τελευταίο βήμα της διαδικασίας θα πρέπει να λάβει χώρα η δημιουργία του τελικού πίνακα γινομένου C, που θα προκύψει από τη συνένωση των υπο-πινάκων **localC** του συνόλου των διεργασιών του συστήματος. Αυτή η διαδικασία θα πραγματοποιηθεί δια της κλήσεως της συνάρτησης **Gather2DMatrix**, η οποία καλείται συλλογικά από όλες τις διεργασίες του communicator **MPI_COMM_WORLD**. Ο πηγαίος κώδικας αυτής της συνάρτησης ακολουθεί στη συνέχεια.

```
int Gather2DMatrix(double ** matrix, int N, double ** local, int dim, int root,
GridInfo * grid)
{
        int flag, rank, loops = N/dim;
        int size, cnt, source, rootCoords[2];
        int counter, index, coords[2], i, j;
        double * tempArray;
        MPI_Status status;

        MPI_Initialized(&flag);
        if(flag == 0)
                return(-1);

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

Η τάξη της τρέχουσας διεργασίας και το συνολικό πλήθος των διεργασιών του προεπιλεγμένου Communicator που περιγράφεται από τη σταθερά **MPI_COMM_WORLD** αποθηκεύονται στις μεταβλητές **rank** και **size** τύπου integer καλώντας τις παραπάνω συναρτήσεις αντίστοιχα.

```
        if((root<0)||(root>=size))
        {
                MPI_Finalize();
                return(-1);
        }
```

Σε περίπτωση που η τάξη της διεργασίας–ρίζα, η οποία θα συλλέξει τους τοπικούς υπο-πίνακες διαστάσεων dimxdim όλων των διεργασιών του συστήματος, δεν έχει έγκυρη τιμή (*if((root<0)||(root>=size)))*, το πρόγραμμα τερματίζει τη λειτουργία του.

```
tempArray =(double *) malloc(dim*dim*sizeof(double));
if(!tempArray)
        return(-1);
```

Μέσω της συνάρτησης **malloc**, δεσμεύεται για το διάνυσμα **tempArray** ο απαραίτητος χώρος στη μνήμη για την αποθήκευση dim*dim αριθμών κινητής υποδιαστολής

```
if(rank==root)
{
        for(counter=0;counter<loops;counter++)
        {
                coords[0]=counter;
                for(index=0;index<loops;index++)
                {
                        coords[1]=index;
                        MPI_Cart_rank(grid->gridComm, coords, &source);
                        if(source==root)
                        {
                                MPI_Cart_coords(grid->gridComm, rank, 2,
                                rootCoords);

                                 for(i=dim*rootCoords[0];
                                i<dim*(rootCoords[0]+1);i++)
                                {
                                        for(j=dim*rootCoords[1];
                                        j<dim*(rootCoords[1]+1);j++)
                                        {
                                                matrix[i][j]=local[i][j];
                                        }
                                }
                        }
                        else
                        {
                                MPI_Recv(tempArray, dim*dim, MPI_DOUBLE,
                                source, 0, MPI_COMM_WORLD, &status);
                                cnt=0;
                                for(i=dim*counter;i<dim*(counter+1);i++)
                                {

for(j=dim*index;j<dim*(index+1);j++)
                                        {
                                                matrix[i][j]=tempArray[cnt];
                                                cnt++;
                                        }
                                }
```

Η διεργασία ρίζα που καθορίζεται από το όρισμα **root** της συνάρτησης, συλλέγει τους τοπικούς υπο-πίνακες διαστάσεων dimxdim όλων των διεργασιών του συστήματος – που περιγράφονται από το όρισμα **localC** – και τοποθετεί τα στοιχεία τους στα κατάλληλα κελιά του τετραγωνικού πίνακα **matrix** τάξεως **N**.

```
                        }
                }
        }
```

```
	}
	else
	{
		cnt=0;
		for(i=0;i<dim;i++)
			for(j=0;j<dim;j++)
			{
				tempArray[cnt]=local[i][j];
				cnt++;
			}
			MPI_Send(tempArray, dim*dim, MPI_DOUBLE, root, 0,
			MPI_COMM_WORLD);
```

Όλες οι διεργασίες του συστήματος στέλνουν στην διεργασία-ρίζα (**root**) τους τοπικούς υπο-πίνακες διαστάσεων dimxdim.

```
	}

	free(tempArray);
```

Η συνάρτηση **free** απελευθερώνει τον χώρο που είχε δεσμευτεί στη μνήμη για το διάνυσμα **tempArray**.

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

Σε πλήρη αναλογία με τη συνάρτηση **Scatter2Dmatrix**, η συνάρτηση **Gather2Dmatrix** θα κληθεί συλλογικά από όλες τις συναρτήσεις του communicator **MPI_COMM_WORLD**.

Μετά την παραλαβή των στοιχείων των πινάκων **localA** και **localB**, η κάθε διεργασία υπολογίζει τα στοιχεία του πίνακα **localC** – που εκφράζει και τη δική της συνεισφορά στη συνολική πράξη πολλαπλασιασμού των πινάκων. Η διαδικασία του υπολογισμού των στοιχείων του πίνακα **localC** θα λάβει χώρα δια της χρήσεως του αλγορίθμου του **Fox** ο οποίος υλοποιείται με τη βοήθεια της ακόλουθης συνάρτησης.

```
int Fox(int matrixOrder, GridInfo * grid, double ** localA, double ** localB,
double ** localC)
{
	int i,sourceRank, x,y, destRank, root, cnt;
	int dim=matrixOrder/grid->gridOrder;
	double ** tempMatrix;
	MPI_Status status;
	double * sendVec, * recvVec;

	MPI_Cart_shift(grid->colComm, 0, 1, &destRank, &sourceRank);
```

Η ερώτηση «ποιοι είναι οι γείτονές μου;» απαντάται με την κλήση της συνάρτησης **MPI_Cart_shift**. Οι τιμές που επιστρέφονται είναι τα ranks, στον communicator **colComm**, των γειτόνων μετατοπισμένων κατά +1/-1 στις δύο διαστάσεις.

```
tempMatrix=Allocate2DMatrix(dim,dim);
sendVec =(double *) malloc(dim*dim*sizeof(double));
recvVec =(double *) malloc(dim*dim*sizeof(double));
```

Μέσω της συνάρτησης **malloc**, δεσμεύεται για τα διανύσματα **sendVec** και **revcVec** ο απαραίτητος χώρος στη μνήμη για την αποθήκευση dim*dim αριθμών κινητής υποδιαστολής.

```
if((!tempMatrix)||(!sendVec)||(!recvVec))
{
        MPI_Finalize();
        return(-1);
}
```

Σε περίπτωση που συμβεί κάποιο σφάλμα κατά τη δημιουργία των διανυσμάτων **tempMatrix**, **sendVec** ή **recvVec**, το πρόγραμμα τερματίζει τη λειτουργία του.

```
for(x=0;x<dim;x++)
        for(y=0;y<dim;y++)
                localC[x][y]=0.0;
```

Όλα τα στοιχεία του πίνακα **localC** αρχικοποιούνται με την τιμή 0.0.

```
for(i=0;i<grid->gridOrder;i++)
{
        root=(grid->currentRow+i)%grid->gridOrder;
        if(root==grid->currentCol)
        {
                cnt=0;
                for(x=0;x<dim;x++)
                {
                        for(y=0;y<dim;y++)
                        {
                                sendVec[cnt]=localA[x][y];
                                cnt++;
                        }
                }
                MPI_Bcast(sendVec, dim*dim, MPI_DOUBLE, root, grid-
                >rowComm);
```

Η εκπομπή του διανύσματος **sendVec** κατά την οριζόντια διεύθυνση της καρτεσιανής τοπολογίας γίνεται από τη συνάρτηση **MPI_Bcast** και δια της χρήσεως του communicator **rowComm.**

```
                LocalMatrixProduct(localA, localB, localC, dim);
```

Η συνάρτηση **LocalMatrixProduct** υπολογίζει το γινόμενο των υπο-πινάκων **localA** και **localB** και το αποθηκεύει στον υπο-πίνακα **localC**.

```
        }
        else
        {
                MPI_Bcast(recvVec, dim*dim, MPI_DOUBLE, root, grid->rowComm);
```

Η αποστολή του διανύσματος **recvVec** σε όλες τις διεργασίες γίνεται από τη συνάρτηση **MPI_Bcast** και δια της χρήσεως του communicator **rowComm.**

```
                for(x=0;x<dim;x++)
                        for(y=0;y<dim;y++)
                                tempMatrix[x][y]=recvVec[x*dim+y];
                LocalMatrixProduct(tempMatrix, localB, localC, dim);
```

Η συνάρτηση **LocalMatrixProduct** υπολογίζει το γινόμενο των υπο-πινάκων **tempMatrix** και **localB** και το αποθηκεύει στον υπο-πίνακα **localC**.

```
        }

        cnt=0;
        for(x=0;x<dim;x++)
                for(y=0;y<dim;y++)
                {
                        sendVec[cnt]=localB[x][y];
                        cnt++;
                }

        MPI_Sendrecv_replace(sendVec, dim*dim, MPI_DOUBLE, destRank, 0, sourceRank, 0, grid->colComm, &status);
```

Η κυκλική εναλλαγή κατά την κατακόρυφη διεύθυνση γίνεται από τη συνάρτηση **MPI_Sendrecv_replace** και δια της χρήσεως του communicator **colComm.**

```
        for(x=0;x<dim;x++)
                for(y=0;y<dim;y++)
                        localB[x][y]=sendVec[x*dim+y];
}

Free2DMatrix(tempMatrix, dim);
```

Μέσω της συνάρτησης **Free2DMatrix** αποδεσμεύεται η μνήμη που είχε δεσμευτεί για την αποθήκευση του πίνακα **tempMatrix**.

```
free(sendVec);
free(recvVec);
```

Οι παραπάνω κλήσεις της συνάρτησης **free** απελευθερώνουν τις θέσεις μνήμης που είχαν δεσμευτεί για τα διανύσματα **sendVec** και **recvVec**.

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

```
int main(int argc, char ** argv)
{
	int i,j,rank, dim, root=0;
	GridInfo grid;
	double ** localA, ** localB, ** localC;
	double ** aMatrix, ** bMatrix, ** cMatrix;
	MPI_Init(&argc, &argv);
```

Μέσω της συνάρτησης **MPI_Init** αρχικοποιείται το περιβάλλον του MPI.

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

Η τάξη της τρέχουσας διεργασίας αποθηκεύεται στην ακέραια μεταβλητή **rank** καλώντας την συνάρτηση **MPI_Comm_rank**.

```
	SetupGrid(&grid);
	dim = ORDER/grid.gridOrder;

	localA=Allocate2DMatrix(dim,dim);
	if(localA==NULL)
		return(-1);
	localB=Allocate2DMatrix(dim,dim);
	if(localB==NULL)
		return(-1);
	localC=Allocate2DMatrix(dim,dim);
	if(localC==NULL)
		return(-1);
```

Χρησιμοποιώντας την συνάρτηση **Allocate2DMatrix** δεσμεύουμε τη μνήμη που απαιτείται για τους πίνακες **localA**, **localB** και **localC** που αντιστοιχούν στους υπο-πίνακες που θα χρησιμοποιηθούν από την κάθε διεργασία.

```
	if(rank==root)
	{
		aMatrix=Allocate2DMatrix(ORDER,ORDER);
		bMatrix=Allocate2DMatrix(ORDER,ORDER);
		cMatrix=Allocate2DMatrix(ORDER,ORDER);
```

Η βασική διεργασία (root) δεσμεύει την αναγκαία μνήμη για τα στοιχεία των πινάκων A, B και C (που περιγράφονται από τις μεταβλητές **aMatrix**, **bMatrix** και **cMatrix**) χρησιμοποιώντας την συνάρτηση **Allocate2DMatrix**.

```
		if((!aMatrix)||(!bMatrix)||(!cMatrix))
		{
			printf("Not enough memory for matrix allocation. Aborting...\n");
			MPI_Finalize();
			return(-1);
		}
```

Σε περίπτωση που συμβεί κάποιο σφάλμα κατά την δέσμευση της απαιτούμενης μνήμης, τυπώνεται στην οθόνη μήνυμα σφάλματος και το πρόγραμμα τερματίζει τη λειτουργία του.

```
for(i=0;i<ORDER;i++)
{
        for(j=0;j<ORDER;j++)
        {
                aMatrix[i][j]=(double)rand()/RAND_MAX;
                bMatrix[i][j]=(double)rand()/RAND_MAX;
                cMatrix[i][j]=0.0;
```

Οι πίνακες **aMatrix**, **bMatrix** αρχικοποιούνται με τυχαίες τιμές ενώ τα στοιχεία του πίνακα **cMatrix** αρχικοποιούνται στη μηδενική τιμή.

```
        }
}

printf("\n\nMatrix A data:\n");
for(i=0;i<ORDER;i++)
{
        for(j=0;j<ORDER;j++)
                printf("%.3f ", aMatrix[i][j]);
        printf("\n");
}

printf("\n\nMatrix B data:\n");
for(i=0;i<ORDER;i++)
{
        for(j=0;j<ORDER;j++)
                printf("%.3f ", bMatrix[i][j]);
        printf("\n");
}
```

Μέσω της εντολής **printf** στους παραπάνω βρόγχους, τυπώνονται στην οθόνη τα στοιχεία των πινάκων Α (**aMatrix**) και Β (**bMatrix**).

```
}

Scatter2DMatrix(aMatrix, ORDER, localA, dim, root, &grid);
Scatter2DMatrix(bMatrix, ORDER, localB, dim, root, &grid);
```

Το αποτέλεσμα της κλήσης της συνάρτησης ***Scatter2DMatrix*** είναι η διασπορά των στοιχείων των πινάκων **aMatrix** και **bMatrix** στις διεργασίες του συστήματος – μετά την επιστροφή της συνάρτησης η κάθε διεργασία έχει παραλάβει τα στοιχεία που της αναλογούν στους πίνακες **localA** και l**ocalB**.

```
Fox(ORDER, &grid, localA, localB, localC);
```

Στην παραπάνω κλήση της συνάρτησης **Fox** το όρισμα **ORDER** αναφέρεται στην τάξη των τετραγωνικών πινάκων Α και Β η οποία στο παράδειγμά μας έχει την τιμή 12. Μετά την επιστροφή αυτής της συνάρτησης η κάθε διεργασία θα έχει υπολογίσει το γινόμενο των δικών της υπο-πινάκων **localA** και **localB** και θα έχει καταχωρήσει τα στοιχεία του στον δικό της υπο-πίνακα **localC**.

```
Gather2DMatrix(cMatrix, ORDER, localC, dim, root, &grid);
```

Μετά την κλήση της συνάρτησης **Gather2DMatrix** οι τιμές του γινομένου των πινάκων A και B θα περιέχονται στα κελιά του πίνακα **cMatrix**, όπως αυτές έχουν υπολογισθεί δια της εφαρμογής του αλγορίθμου του Fox.

```
if(rank==root)
{
        printf("\n\nMatrix C data:\n");
        for(i=0;i<ORDER;i++)
        {
                for(j=0;j<ORDER;j++)
                        printf("%.3f ", cMatrix[i][j]);
                printf("\n");
        }
```

Μέσω του παραπάνω μπλοκ εντολών, η βασική διεργασία (root) του συστήματος τυπώνει στην οθόνη τα στοιχεία του πίνακα C.

```
}

Free2DMatrix(localA, dim);
Free2DMatrix(localB, dim);
Free2DMatrix(localC, dim);
```

Με χρήση της εντολής **Free2DMatrix** αποδεσμεύεται η μνήμη που προηγουμένως είχε δεσμευτεί για τους πίνακες **localA**, **localB** και **localC**.

```
MPI_Barrier(MPI_COMM_WORLD);
```

Η συνάρτηση **MPI_Barrier** μπλοκάρει την διεργασία που την καλεί μέχρι να κληθεί από όλες τις διεργασίες που ανήκουν στον communicator **MPI_COMM_WORLD**.

```
if(rank==root)
{
        Free2DMatrix(aMatrix, ORDER);
        Free2DMatrix(bMatrix, ORDER);
        Free2DMatrix(cMatrix, ORDER);
```

Με χρήση της εντολής **Free2DMatrix** αποδεσμεύεται η μνήμη που προηγουμένως είχε δεσμευτεί για τους πίνακες **aMatrix**, **bMatrix** και **cMatrix**.

```
}

MPI_Comm_free(&grid.gridComm);
MPI_Comm_free(&grid.rowComm);
MPI_Comm_free(&grid.colComm);
```

Μέσω της συνάρτησης **MPI_Comm_free** αποδεσμεύονται οι communicators **grid.gridComm**, **grid.rowComm** *και* **grid.colComm**.

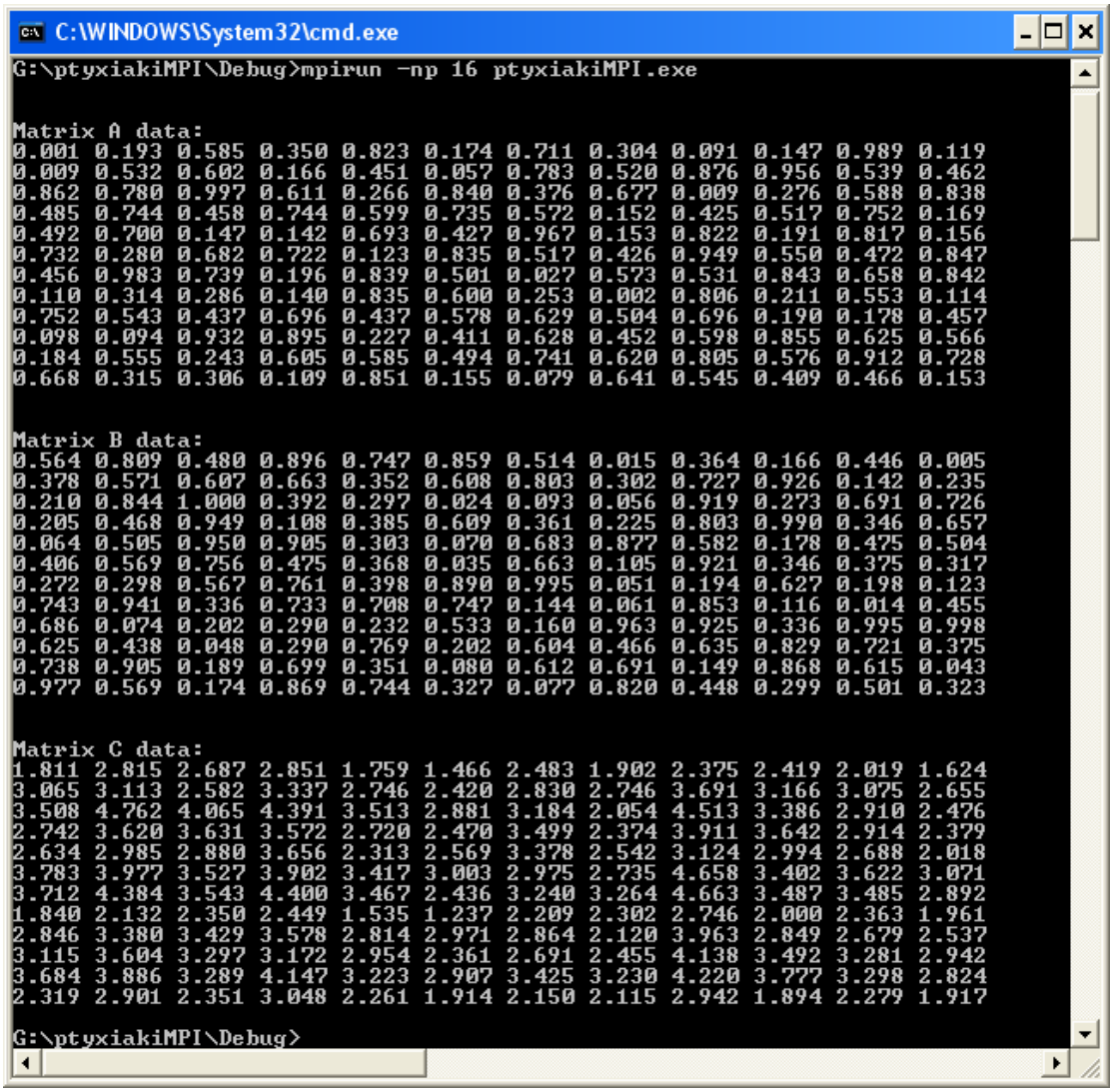
```
        MPI_Finalize();
```

Με την κλήση της **MPI_Finalize** απελευθερώνεται η μνήμη που δεσμεύεται από τις δομές δεδομένων του προτύπου MPI και γενικότερα τερματίζεται η λειτουργία του.

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

7.1.2. Παράδειγμα εκτέλεσης του προγράμματος

Παρακάτω παρουσιάζεται το πρόγραμμα κατά την εκτέλεσή του από ένα σύστημα 16 διεργασιών.



ΚΕΦΑΛΑΙΟ 8

8. Εγκατάσταση του προτύπου MPI

Για τις ανάγκες της παρούσας εργασίας, χρησιμοποιήθηκε η έκδοση MPICH 1.2.5 για Windows NT/2000, η οποία διατίθεται ελεύθερα απο τη διεύθυνση :
http://www.neural.uom.gr/Documents/PDP/mpich.nt.1.2.5.exe.

Μετά την εκτέλεση του αρχείου setup "mpich.nt.1.2.5.exe" και την ολοκλήρωση της εγκατάστασης, τα στοιχεία που εγκαθίστανται είναι τα εξής:

- Run time βιβλιοθήκες dll
- Δύο MPI launchers
- SDK (software development kit)
- SDK.gcc

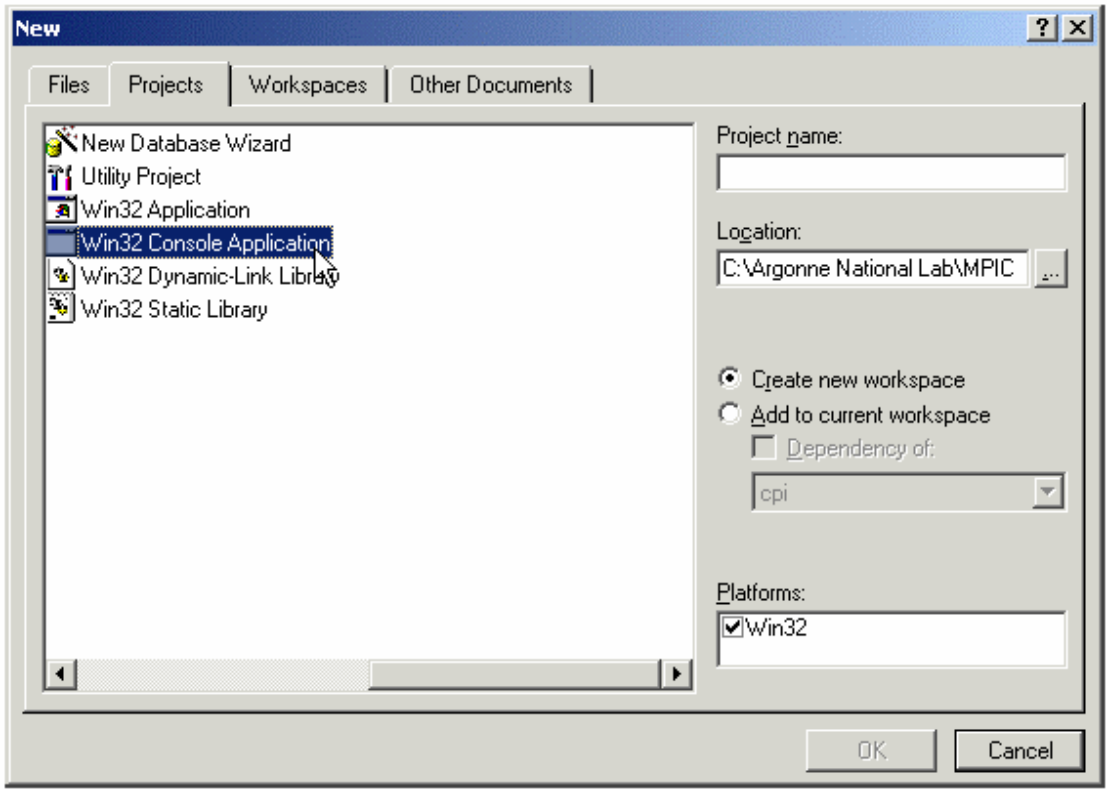
Οι βιβλιοθήκες και οι launchers είναι απαραίτητα στοιχεία και πρέπει να επιλεχθούν για εγκατάσταση όταν αυτό ζητηθεί. Το SDK περιέχει όλες τις βιβλιοθήκες και τα include αρχεία που χρειάζονται για τη μεταγλώττιση προγραμμάτων MPI γραμμένα σε Visual Fortran 6 ή Visual C++ 6. Αντίθετα, το SDK.gcc περιέχει τα αντίστοιχα αρχεία για μεταγλώττιση με gcc.

8.1. Ρυθμίσεις στο Microsoft Visual Studio

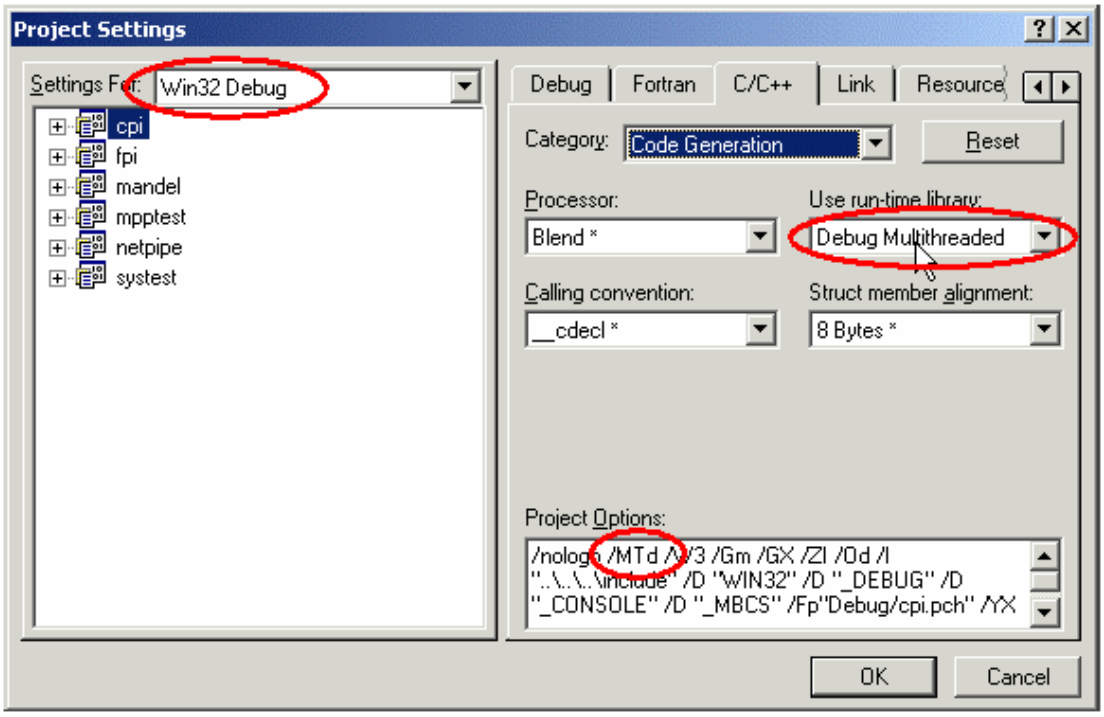
Προκειμένου να μπορέσουμε να χρησιμοποιήσουμε τις βιβλιοθήκες του MPI και να μεταγλωττίσουμε σωστά το πρόγραμμα μας, είναι απαραίτητη μια σειρά ενεργειών.

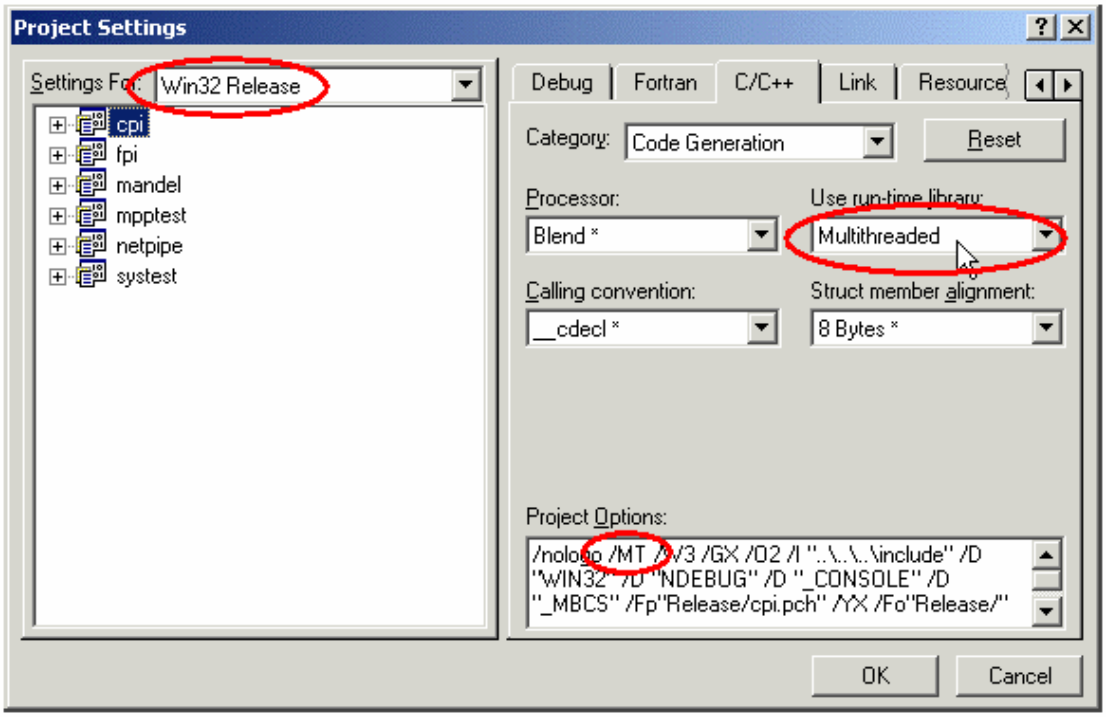
Στην Visual C++ 6, ακολουθούμε τα παρακάτω βήματα :

1. Δημιουργούμε ένα νέο project, το οποίο θα «στεγάσει» τα αρχεία του προγράμματος μας. Επιλέγουμε Win32 Console Application.

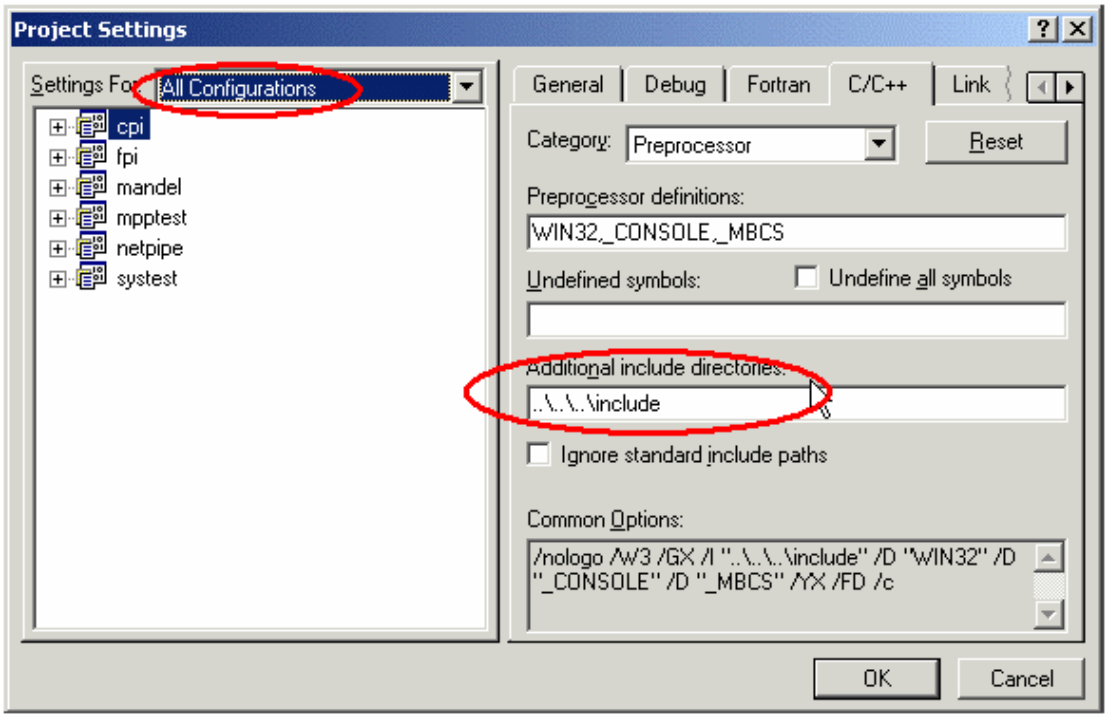


2. Πηγαίνουμε στα Project settings.
3. Αλλάζουμε τις επιλογές για την παραγωγή κώδικα, σε Multithreaded και για τις debug αλλά και για τις release εκδόσεις, όπως φαίνεται στα δύο παρακάτω σχήματα.

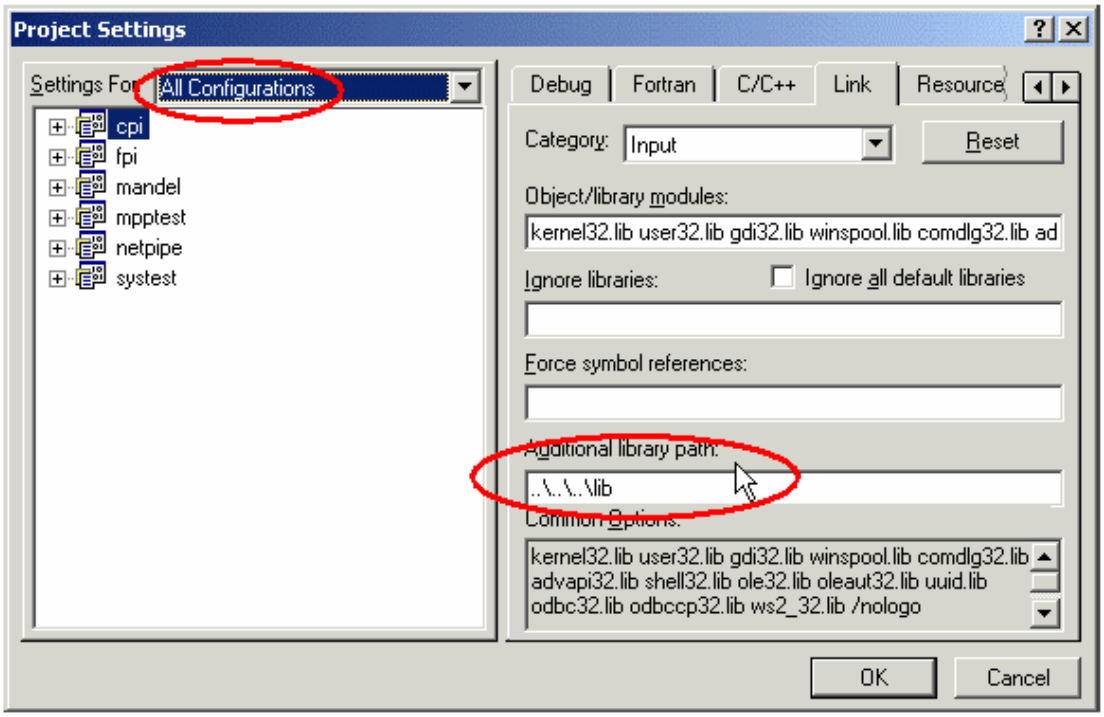




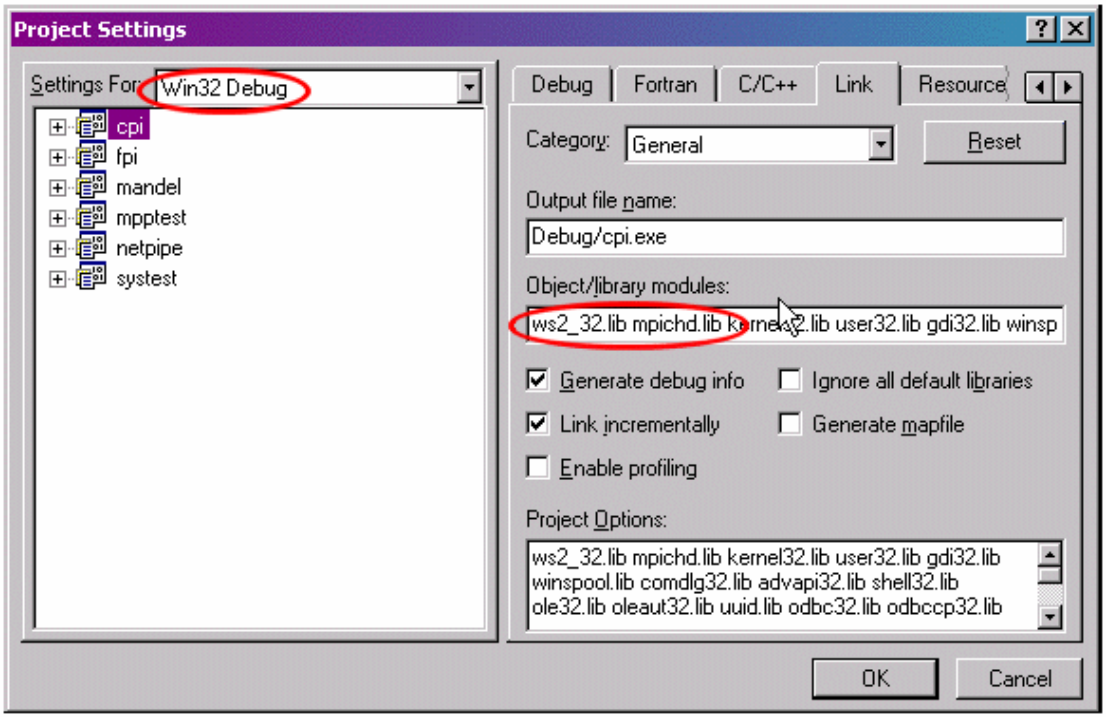
4. Αλλάζουμε την διαδρομή για τα include αρχεία σε <Φάκελος εγκατάστασης MPI>\SDK\include



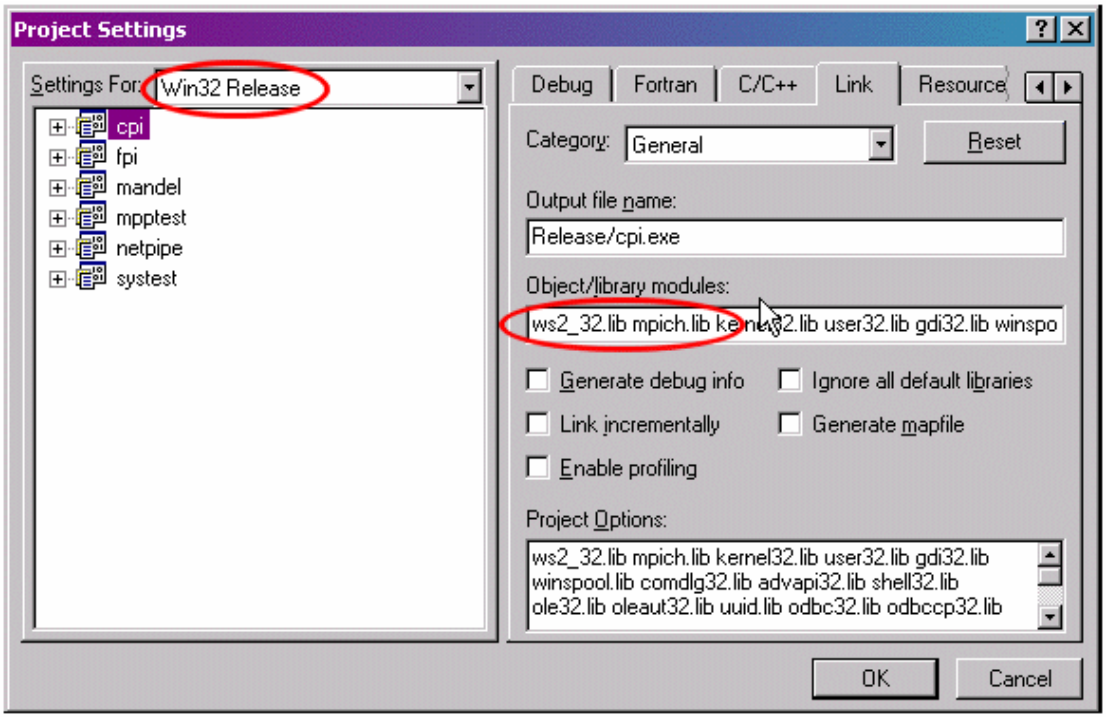
5. Αλλάζουμε την διαδρομή για τα lib αρχεία σε <Φάκελος εγκατάστασης MPI>\SDK\lib



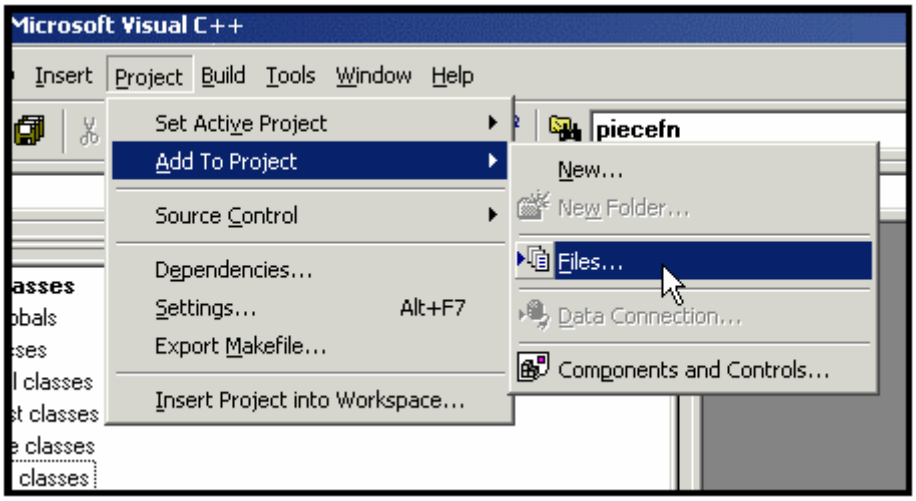
6. Προσθέτουμε το αρχείο ws2_32.lib και το mpichd.lib για τις debug εκδόσεις.



7. Προσθέτουμε το αρχείο ws2_32.lib και το mpich.lib για τις release εκδόσεις.



8. Κλείνουμε τις ρυθμίσεις του project, και τώρα πλέον, μπορούμε να προσθέσουμε τα αρχεία με τον κώδικα μας στο συγκεκριμένο project.



8.2. Εκτέλεση προγραμμάτων MPI

Το πρότυπο MPICH, περιλαμβάνει δύο launchers :
- έναν για την γραμμή εντολών (MPIRun) και
- έναν με γραφικό περιβάλλον (GUIMPIRun).

Πριν εκτελέσουμε οποιοδήποτε πρόγραμμα κάνοντας χρήση ενός launcher, πρέπει να ορίσουμε κάποιες παραμέτρους, όπως τα ονόματα των υπολογιστών στους οποίους θα τρέξει το πρόγραμμα, πόσες διεργασίες θα εκτελεστούν συνολικά αλλά και σε κάθε κόμβο ξεχωριστά κτλ. Αυτό μπορεί να γίνει με την χρήση κάποιου configuration αρχείου εάν πρόκειται να χρησιμοποιήσουμε τον MPIRun launcher, ή με την χρήση ενός γραφικού περιβάλλοντος, τον MPIConfig, εάν πρόκειται να χρησιμοποιήσουμε οποιονδήποτε απο τους δύο launchers.

MPIRun

Γραμμή εντολών

• MPIRun [-παράμετροι] <αρχείο παραμέτρων> [args...]
• MPIRun -np <αριθμός διεργασιών> [-παράμετροι] <εκτελέσιμο αρχείο> [args...]

Αρχείο παραμέτρων

Το αρχείο παραμέτρων μπορεί να περιέχει τις εξής πληροφορίες :
exe c:\somepath\myapp.exe
ή \\host\share\somepath\myapp.exe
[args arg1 arg2 arg3...]
[env VAR1=VAL1|VAR2=VAL2|...|VARn=VALn]
[dir drive:\some\path]
[map drive:\\host\share]

hosts
hostA <αριθμός διεργασιών> [path\myapp.exe]
hostB <αριθμός διεργασιών>
[\\host\share\somepath\myapp2.exe]
hostC <αριθμός διεργασιών>
...

Στην πρώτη γραμμή, μετά την λέξη exe, ορίζουμε σε ποια διαδρομή σε κάθε υπολογιστικό κόμβο ή σε ποια κοινή τοποθεσία στο δίκτυο βρίσκεται το εκτελέσιμο αρχείο. Μετά την γραμμή hosts ορίζουμε σε ποιούς κόμβους και πόσες διεργασίες θα εκτελεστούν.

Παράμετροι

-np <αριθμός διεργασιών>
Εκτελούνται <αριθμός διεργασιών> διεργασίες, αρχίζοντας από τον τρέχοντα υπολογιστικό κόμβο και συνεχίζοντας, μια σε κάθε έναν απο τους υπολογιστές που έχουμε ορίσει, μέχρις ότου να εξαντληθούν όλες οι διεργασίες. Εάν ο αριθμός των διεργασιών είναι μεγαλύτερος από τον αριθμό των υπολογιστών, τότε η ανάθεση των διεργασιών συνεχίζεται με κυκλικό τρόπο.

-machinefile <όνομα αρχείου>
Με αυτήν την παράμετρο οι διεργασίες ανατίθενται σε συγκεκριμένους υπολογιστές, τους οποίους ορίζουμε μέσα στο αρχείο <όνομα αρχείου>. Γράφουμε ένα υπολογιστή ανά γραμμή, ενώ μπορούμε να παραθέσουμε δίπλα σε κάθε κόμβο και ένα νούμερο που θα δηλώνει πόσες διεργασίες θα εκτελεστούν στον συγκεκριμένο υπολογιστή. Με αυτόν τον τρόπο, μπορούμε να εκμεταλλευτούμε τυχόν υπολογιστές με πάνω από μια επεξεργαστικές μονάδες καλύτερα.

-localonly
Με αυτήν την παράμετρο, όλες οι διεργασίες εκτελούνται τοπικά, στον τρέχοντα υπολογιστή, κάνοντας χρήση του μηχανισμού κοινής μνήμης.

-localonly -tcp
Προσθέτοντας την επιλογή –tcp, αναγκάζουμε τον launcher να χρησιμοποιήσει sockets αντί για την μέθοδο της κοινής μνήμης.

-env "var1=val1|var2=val2|var3=val3|...varn=valn"
Με αυτόν τον τρόπο θέτουμε κάποιες μεταβλητές περιβάλλοντος.

-logon
Ο launcher ζητάει username και κωδικό ενός υπάρχοντος λογαριασμού.

-map drive:\\host\share
Με αυτήν την επιλογή, θα γίνει mapping μιας διαδρομής σε δίσκο, στους υπολογιστές στους οποίους θα εκτελεστούν οι διεργασίες.

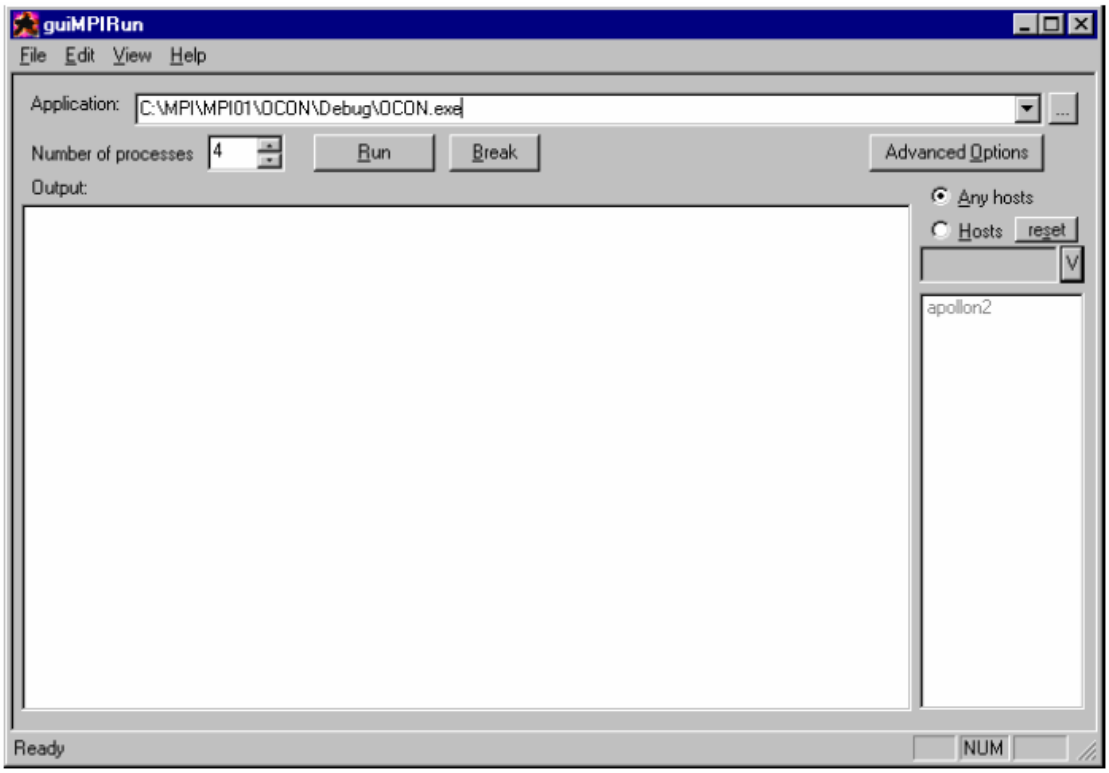
-dir drive:\some\path
Ορίζουμε τον φάκελο εργασιών (working directory). Εάν δεν οριστεί, τότε χρησιμοποιείται ο τρέχων κατάλογος.

-hosts n host1 host2... hostn
-hosts n host1 m1 host2 m2... hostn mn
Ορίζουμε σε ποιους υπολογιστές θα εκτελεστούν οι διεργασίες. Στην δεύτερη περίπτωση, ορίζουμε και τον αριθμό των διεργασιών σε κάθε κόμβο.

-pwdfile <όνομα αρχείου>
Δηλώνουμε ποιο αρχείο περιέχει το username και τον κωδικό, με τους οποίους θα εκτελεστεί η εφαρμογή. Η πρώτη γραμμή περιέχει το username, ενώ η δεύτερη τον κωδικό.

GUIMPIRun

Το περιβάλλον του GUIMPIRun προσφέρει έναν πιο εύχρηστο τρόπο εκτέλεσης των προγραμμάτων MPI. Απο την κεντρική του οθόνη, μπορούμε να επιλέξουμε τους υπολογιστές στους οποίους θα ανατεθούν διεργασίες, το εκτελέσιμο αρχείο, πόσες διεργασίες θα εκτελεστούν κτλ. Γενικά οι επιλογές είναι ακριβώς οι ίδιες με αυτές του μη γραφικού MPIRun launcher.



ΒΙΒΛΙΟΓΡΑΦΙΑ

- Ησαιάδης Σταύρος, *«Εισαγωγή στο MPI»*, http://www.neural.uom.gr/

- Μάργαρης Αθανάσιος, *«Παράλληλα και κατανεμημένα συστήματα, Το περιβάλλον προγραμματισμού MPI»*, http://www.neural.uom.gr/

- Σταυρουλάκης Γιώργος, *«Βασικές Αρχές Προγραμματισμού σε Παράλληλο Περιβάλλον»,* Εθνικό Μετσόβιο Πολυτεχνείο, Τμήμα Χημικών Μηχανικών, Νοέμβριος 2004, http://users.civil.ntua.gr/papadrakakis/files/Multiprocessing.pdf

- Blaise Barney, *«Introduction to Parallel Computing»,* Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/

- Karniadakis Em George and Robert M. Kirby II, *«Parallel Scientific Computing in C++ and MPI, A seamless approach to parallel algorithms and their implementation»,* Cambridge University Press

- Margaris Athanasios, Souravlas Stavros, Roumeliotis Manos *«Parallel Implementations of the Jacobi, Linear Algebraic Systems Solver»*, Department of Applied Informatics, University of Macedonia

- Yukiya Aoyama, Jun Nakano, *«RS/6000 SP: Practical MPI Programming»,* International Technical Support Organization, August 1999, www.redbooks.ibm.com