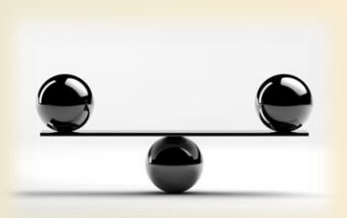
Μεταπτυχιακό Δίπλωμα Ειδίκευσης

Μηχανική Λογισμικού για Διαδικτυακές και Φορητές Εφαρμογές

Μεταπτυχιακή εργασία της

Μαρίας Δρούγκα



Συγκριτική Μελέτη των Γλωσσών Προγραμματισμού C και Python με χρήση της MPI ως προς την ταχύτητα

Εισηγητής

Δρ. Ηλίας Κ. Σάββας

Λάρισα, 29 Ιανουαρίου 2020

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συγκριτική Μελέτη των Γλωσσών Προγραμματισμού C και Python με χρήση της MPI ως προς την ταχύτητα

Μαρία Δρούγκα

Α.Μ. 7418004

Εισηγητής:

Δρ. Ηλίας Κ. Σάββας

Εξεταστική επιτροπή:

1. Η. Σάββας

2. Γ. Γκαράνη

3. Β. Κουτσονικόλα

Ημερομηνία εξέτασης: 29/01/2020

"An approximate answer to the right question is worth a great deal more than a precise answer to the wrong question"-
John Tukey

Στον σύζυγό μου Νικόλαο και
στα παιδιά μου
Στέλλα και Γιώργο

Ευχαριστίες

Και κάπως έτσι έκλεισε ένας κύκλος για μένα… ένας κύκλος που άνοιξε δύσκολα κι έκλεισε με πολλά ποικίλα συναισθήματα. Ήταν δύσκολο για μένα να βρεθώ μετά από αρκετά χρόνια ξανά σε θρανία και με το άγχος των εξετάσεων. Πήρα αυτή την απόφαση με πολλά ερωτηματικά για το αν θα μπορούσα να τα βγάλω εις πέρας λόγω οικογενειακών υποχρεώσεων και φυσικά δεν το μετάνιωσα.

Σύμμαχοι σε όλο αυτό ήταν η οικογένειά μου που την ευχαριστώ για την υπομονή που μου έδειξε όλο αυτό το διάστημα των δυο χρόνων και το κουράγιο που μου έδωσε να συνεχίσω σε αυτό τον δύσκολο αγώνα.

Επίσης θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου, Ηλία Σάββα, για τη συνεχή υποστήριξη, τις σημαντικές παρατηρήσεις του, την υπομονή και την εμπιστοσύνη που μου έδειξε προκειμένου να ολοκληρωθεί αυτή η εργασία. Ως υπεύθυνος του μεταπτυχιακού αυτού προγράμματος, χάρη στην πνευματική του καλλιέργεια, στην επιστημονική του κατάρτιση, στην ευγένεια και στην ικανότητα επικοινωνίας που τον χαρακτηρίζει, συνέβαλε τα μέγιστα για την ολοκλήρωση των σπουδών μου.

Ευχαριστίες οφείλω τέλος και στο σύνολο των καθηγητών του μεταπτυχιακού προγράμματος καθώς μου πρόσφεραν τα κατάλληλα εφόδια κατά την διάρκεια των μαθημάτων για την απόκτηση του μεταπτυχιακού διπλώματος.

Πρόλογος

Η πιο συνηθισμένη τακτική προγραμματισμού που συναντάται σε καθημερινή βάση είναι ο ***σειριακός προγραμματισμός***. Σε αυτήν την περίπτωση το πρόγραμμα εκτελείται σειριακά, δηλαδή η μια εντολή μετά την άλλη. Σε κάποιες περιπτώσεις όμως αυτό δεν είναι αρκετό διότι ο περιορισμός στο χρόνο ή στους υπολογιστικούς πόρους μπορεί να είναι απαγορευτικός για τους στόχους μας. Έτσι, δημιουργείται η ανάγκη για μια άλλη τακτική που ακούει στο όνομα ***παράλληλος προγραμματισμός***. Με αυτόν τον τρόπο μπορούμε να εκμεταλλευτούμε την ύπαρξη πολλαπλών επεξεργαστών και να αυξήσουμε την υπολογιστική επίδοση μειώνοντας ταυτόχρονα το χρόνο εκτέλεσης.

Στη συγκεκριμένη εργασία θα μιλήσουμε για το MPI που αποτελεί ένα βασικό μοντέλο παράλληλου προγραμματισμού και χρησιμοποιείται κυρίως σε συστήματα κατανεμημένης μνήμης. Στα συστήματα αυτά ο κάθε επεξεργαστής έχει πρόσβαση στη δική του μνήμη, δηλαδή σε συγκεκριμένα δεδομένα.

Αν και οι πιο κατάλληλες γλώσσες για παράλληλο υπολογισμό είναι η C ή η Fortran, πολλοί ερευνητές προτιμούν τις γλώσσες που βασίζονται σε array (όπως η Python ή η Matlab) πάνω από C και Fortran για την ευκολία χρήσης τους. Συγκεκριμένα, η Python έχει γίνει όλο και πιο δημοφιλής.

Σκοπός αυτής της εργασίας είναι να κάνουμε μια σύγκριση ως προς την ταχύτητα εκτέλεσης βασικών παράλληλων προγραμμάτων με χρήση της MPI στις δύο γλώσσες C και Python. Η σύγκριση θα αφορά αριθμητικά δεδομένα και θα γίνει και γραφική αναπαράσταση αυτών.

Λέξεις κλειδιά: *σειριακός προγραμματισμός, παράλληλος προγραμματισμός, κατανεμημένη μνήμη, MPI, C, Python, ταχύτητα, σύγκριση*

Abstract

The most common programming tactic encountered on a daily basis is **serial programming**. In this case the program is executed sequentially, that is, one command after another. In some cases, however, this is not enough because time or computing resources can be prohibitive for our purposes. Thus, the need arises for another tactic that hears in the name of parallel programming. This way we can take advantage of multiple processors and increase computational performance while reducing runtime.

In this thesis we will talk about MPI, which is a basic **parallel programming** model and is mainly used in distributed memory systems. In these systems each processor has access to its own memory, that is, to specific data.

Although the most suitable computational languages are C or Fortran, many researchers prefer array-based languages (such as Python or Matlab) over C and Fortran for ease of use. In particular, Python has become increasingly popular.

The purpose of this thesis is to compare the running speed of basic parallel programs using MPI in both C and Python languages. The comparison will involve numerical data and a graphical representation of them.

Keywords: *serial programming, parallel programming, shared memory, MPI, C, Python, speed, comparison*

Περιεχόμενα

Κατάλογος πινάκων

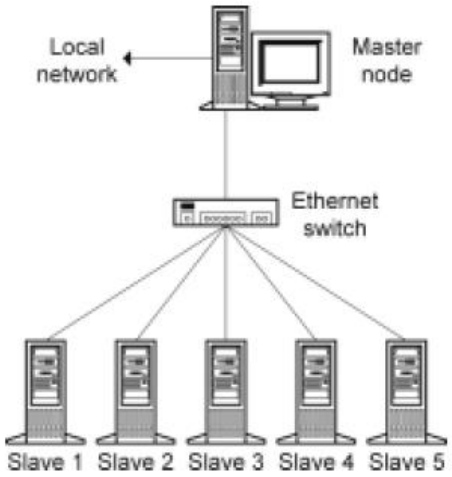
Κατάλογος σχημάτων

Κεφάλαιο 1

MPI με C

1.1 Εισαγωγή στον παράλληλο προγραμματισμό

Η πιο γνωστή τακτική προγραμματισμού που συναντάμε σε καθημερινή βάση είναι ο σειριακός προγραμματισμός. Λέγοντας ότι ένα πρόγραμμα εκτελείται σειριακά, εννοούμε ότι εκτελείται η μια εντολή μετά την άλλη. Μερικές φορές όμως αυτό δεν αρκεί επειδή οι περιορισμοί σε χρόνο ή σε υπολογιστικούς πόρους μπορεί να είναι απαγορευτικοί για τους στόχους μας. Οδηγούμαστε έτσι σε μια άλλη τακτική προγραμματισμού που ονομάζεται **παράλληλος προγραμματισμός**.[4] Μπορούμε λοιπόν να εκμεταλλευτούμε την ύπαρξη πολλαπλών επεξεργαστών και να αυξήσουμε την υπολογιστική επίδοση μειώνοντας ταυτόχρονα το χρόνο εκτέλεσης. Δημιουργούνται έτσι οι επονομαζόμενες συστοιχίες υπολογιστών (computer clusters). Πρόκειται για δίκτυα υπολογιστών καθημερινής χρήσης που επιτρέπουν σε απαιτητικές εφαρμογές να εκτελεστούν παράλληλα. Τα δίκτυα αυτά χρησιμοποιούν εξελιγμένους μηχανισμούς προκειμένου να διαμοιράσουν τους πόρους όλων των διαθέσιμων κόμβων.



Σχήμα 1: Αρχιτεκτονική συστοιχιών υπολογιστών[6]

Στο παραπάνω σχήμα παρουσιάζεται η αρχιτεκτονική των συστοιχιών υπολογιστών. Μια τυπική συστοιχία περιλαμβάνει έναν αριθμό υπολογιστών συνδεδεμένο μεταξύ τους κι έναν κεντρικό υπολογιστή ο οποίος αναλαμβάνει τον συντονισμό τους. Σε κάθε έναν από τους υπολογιστές αυτούς υπάρχει εγκατεστημένο ειδικό λογισμικό που χρησιμεύει στη διασύνδεση των κόμβων μεταξύ τους και ονομάζεται ενδιάμεσο στρώμα (middleware).

Εν αντιθέσει με το σειριακό μοντέλο προγραμματισμού όπου χρησιμοποιείται ένας και μοναδικός επεξεργαστής όπου μέσω διαδικασιών ανάγνωσης κι εγγραφής επικοινωνεί συνεχώς με την κεντρική μνήμη του συστήματος, στο αντίστοιχο παράλληλο μοντέλο χρησιμοποιούνται πολλοί διαφορετικοί επεξεργαστές όπου κάθε ένας χρησιμοποιεί τη δική του περιοχή μνήμης καθώς και τις δικές του μεταβλητές προγράμματος. Αυτό σημαίνει πως οι επεξεργαστές δε διαθέτουν κάποιο φυσικό χώρο διευθύνσεων μνήμης που να είναι κοινός και να τον μοιράζονται, και επομένως οι δραστηριότητες μιας διεργασίας που εκτελούνται σε ένα υπολογιστή, δε γίνονται ορατές από τις υπόλοιπες διεργασίες του συστήματος. Επιπλέον αν σε μία διεργασία είναι απαραίτητη μια μεταβλητή που όμως ανήκει σε μία άλλη διεργασία, τότε επειδή αυτή η μεταβλητή δεν ανήκει από κοινού και στις δύο διεργασίες, θα πρέπει να σταλεί από τη μια διεργασία στην άλλη μέσω κάποιου μηνύματος (message). [3]

Σημαντικό θέμα στον προγραμματισμό πολλαπλών πυρήνων είναι η διαμοίραση της εργασίας και ο συγχρονισμός των πυρήνων. Τα δύο βασικά μοντέλα παράλληλου προγραμματισμού που μας ενδιαφέρουν είναι:

- Message-Passing Programming (**MPI**)
- Shared-Memory Programming (**OpenMP**)

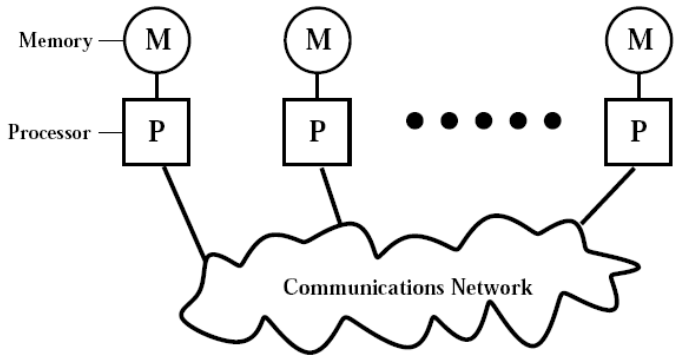
Τα παράλληλα συστήματα διακρίνονται στα κοινής μνήμης και στα κατανεμημένης μνήμης. Στα μεν πρώτα οι επεξεργαστές έχουν πρόσβαση σε μια κοινή μνήμη, δηλαδή σε όλα τα δεδομένα, στα δε δεύτερα ο κάθε επεξεργαστής έχει πρόσβαση στη δική του μνήμη, δηλαδή σε συγκεκριμένα δεδομένα. [4]

Το μοντέλο μεταβίβασης μηνυμάτων(MPI) είναι μία τεχνική παράλληλου προγραμματισμού που χρησιμοποιείται σε συστοιχίες υπολογιστών. Σε αυτή την περίπτωση ο κάθε συμμετέχων κόμβος στη συστοιχία έχει ξεχωριστή μνήμη και η μεταξύ τους επικοινωνία γίνεται με ανταλλαγή μηνυμάτων. Επειδή ο προγραμματιστής θα πρέπει να προσδιορίσει ακριβώς το περιεχόμενο, τους αποστολείς και παραλήπτες των διακινούμενων μηνυμάτων καθώς και τα σημεία εκείνα στα οποία διακόπτονται προσωρινά οι υπολογισμοί και αρχίζουν οι επικοινωνίες, θεωρείται από πολλούς ότι πρόκειται για ένα από τα σχετικά δύσκολα μοντέλα παράλληλου προγραμματισμού. Και όλα αυτά επειδή πέρα από τον ρητό

διαμοιρασμό των υπολογιστικών εργασιών, ο προγραμματιστής πρέπει να κάνει επιπλέον ρητό τον διαμοιρασμό των δεδομένων του προγράμματος. Εντούτοις, πρόκειται για φθηνή λύση στον παράλληλο υπολογισμό.

Βασική μέθοδο για τον προγραμματισμό διαφόρων πολυεπεξεργαστών κατανεμημένης μνήμης συνιστά η μεταβίβαση μηνυμάτων. Από παλιά γνωρίζουμε ότι οι κατασκευαστές φρόντισαν να υπάρχει στα συστήματα τους ξεχωριστός μεταφραστής της γλώσσας C(που μας ενδιαφέρει εδώ) με κατάλληλες επεκτάσεις για τον προγραμματισμό με μεταβίβαση μηνυμάτων. Τα προγράμματα ήταν γραμμένα σε πηγαίο κώδικα και η μεταφορά τους σε άλλον υπολογιστή κρίνονταν δύσκολη και χρονοβόρα. Ο συνδυασμός μάλιστα με την προσπάθεια να αναδειχθούν τα τοπικά δίκτυα ως μια εναλλακτική λύση οικονομικών παράλληλων υπολογιστών, συντέλεσε στον ασπασμό προτύπων, καθολικά αποδεκτών. Ανάμεσα σε αυτά τα πρότυπα ξεχώρισε η ύπαρξη του **MPI** (Message Passing Interface). Το **MPI** αποτελεί πλέον μονόδρομο για προγραμματισμό με μεταβίβαση μηνυμάτων τόσο σε παράλληλους υπολογιστές κατανεμημένης μνήμης, όσο και σε τοπικές συστοιχίες αποτελούμενες από σταθμούς εργασίας. [8]

Το **MPI** έγινε εξαιρετικά αρεστό με την πάροδο του χρόνου λόγω της ευκολίας χρήσης του αλλά και λόγω του ότι μπορούσαν να το υποστηρίξουν περισσότερες πλατφόρμες. Τα προγράμματα που γράφονται με αυτό τον τρόπο μπορούν να εκτελούνται σε διαφορετικές αρχιτεκτονικές υπολογιστών, ενώ ο εκτελέσιμος κώδικας που παράγεται είναι αρκετά αποδοτικός.[7]

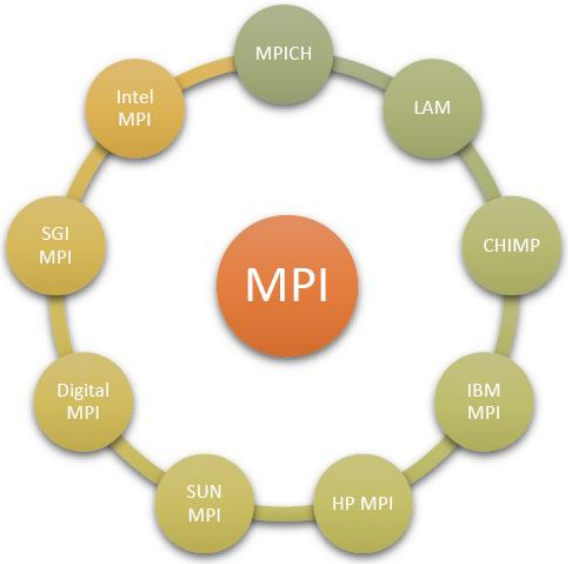


Σχήμα 2: Μοντέλο Μεταβίβασης Μηνυμάτων[6]

1.2 Ιστορικά το πρότυπο MPI

Το πρότυπο MPI αποτελείται από ένα σύνολο συναρτήσεων που μπορεί να χρησιμοποιηθεί σε προγράμματα που είναι γραμμένα σε γλώσσα C, C++ ή Fortran. Τα προγράμματα αυτά επιτρέπουν τη δημιουργία και χρήση ενός συνόλου διεργασιών που εκτελούνται παράλληλα στον ίδιο ή σε διαφορετικούς υπολογιστές και επικοινωνούν μεταξύ τους χρησιμοποιώντας το μοντέλο ανταλλαγής μηνυμάτων. [2,5]

Το καλοκαίρι του 1994 κυκλοφόρησε η έκδοση 1 αυτού του προτύπου (αποκαλείται με το όνομα MPI-1). Από την κυκλοφορία της, η προδιαγραφή MPI έχει γίνει το κορυφαίο πρότυπο για τις βιβλιοθήκες που μεταδίδουν μηνύματα για τους παράλληλους υπολογιστές. Όπως φαίνεται και στο διπλανό σχήμα, πολλές εταιρίες ανέπτυξαν μια διαφορετική έκδοση για το MPI. Ένας σημαντικός λόγος για την ταχεία υιοθέτηση του MPI ήταν η εκπροσώπηση στο MPI Forum, το οποίο σχεδίασε το MPI, από όλα τα τμήματα της παράλληλης υπολογιστικής κοινότητας: προμηθευτές, συγγραφείς βιβλιοθηκών και επιστήμονες εφαρμογών.



Σχήμα 3: Διάφορες υλοποιήσεις της αρχιτεκτονικής MPI[3]

Η προσπάθεια τυποποίησης MPI-1 περιελάμβανε πάνω από ογδόντα άτομα από σαράντα οργανισμούς, κυρίως από τις Ηνωμένες Πολιτείες και την Ευρώπη. Οι περισσότεροι από τους σημαντικότερους προμηθευτές παράλληλων υπολογιστών τότε συμμετείχαν στην MPI, μαζί με ερευνητές από πανεπιστήμια, κρατικά εργαστήρια και τη βιομηχανία. Η διαδικασία τυποποίησης ξεκίνησε με το Workshop για τα πρότυπα για τη μετάδοση μηνυμάτων σε περιβάλλον κατανεμημένης μνήμης, το οποίο χρηματοδοτήθηκε από το Κέντρο Έρευνας για τον Παράλληλο Υπολογισμό, που πραγματοποιήθηκε στις 29-30 Απριλίου 1992 στο Williamsburg της Βιρτζίνια. Ένα πρόχειρο σχέδιο πρότασης συστήθηκε από τους Dongarra, Hempel, Hey και Walker τον Νοέμβριο του 1992 και μια βελτιωμένη έκδοση ολοκληρώθηκε τον Φεβρουάριο του 1993.

Τον Νοέμβριο του 1992 οργανώθηκε στη Μινεάπολη σύσκεψη από την ομάδα εργασίας του MPI, που είχε ως αποτέλεσμα να δοθεί στη διαδικασία τυποποίησης πιο σημαίνουσα αξία. Στους πρώτους εννέα μήνες του 1993, η ομάδα εργασίας MPI οργάνωνε συναντήσεις κάθε έξι εβδομάδες. Το πρόχειρο σχέδιο του προτύπου MPI προβλήθηκε στη συνεδρίαση του Supercomputing '93 τον Νοέμβριο του 1993. Τελικά, η πρώτη έκδοση του MPI δημοσιεύτηκε το Μάιο του 1994, μετά από κάποια δημόσια σχόλια που επέφεραν αλλαγές στο MPI.

Πολύ σημαντική υπήρξε και η δημιουργία του φόρουμ MPI που ουσιαστικά δημιουργήθηκε από τις συναντήσεις αυτές καθώς και με τη συζήτηση μέσω ηλεκτρονικού ταχυδρομείου. Η συμμετοχή στο φόρουμ ήταν ανοιχτή σε όλα τα μέλη της κοινότητας υπολογιστών υψηλών επιδόσεων.

Η έκδοση 1 του MPI δεν αναφέρθηκε σε ορισμένα θέματα που ήταν σημαντικά για τον παράλληλο υπολογισμό. Ο λόγος ήταν ότι το Forum επιθυμούσε να ολοκληρώσει το έργο του σε περιορισμένο χρονικό διάστημα. Κατά την άνοιξη του 1995, το φόρουμ συγκαλέστηκε ξανά προκειμένου να αντιμετωπίσει αυτά τα θέματα όπως και άλλα που προέκυψαν από την εφαρμογή και τη χρήση του MPI-1. Το "νέο" Φόρουμ MPI αποτελούνταν πλέον και από νέους συμμετέχοντες εκτός από το MPI-1. Ακολούθησε την ίδια ανοικτή διαδικασία όπως προηγουμένως, ενθαρρύνοντας όποιον επιθυμούσε να συμμετάσχει σε συναντήσεις και σε συζητήσεις ηλεκτρονικού ταχυδρομείου που λάμβαναν χώρα μεταξύ των συνεδριάσεων.

Τον Ιούλιο του 1997 το MPI Forum ολοκλήρωσε τον ορισμό του MPI-2. Περιελάμβανε σημαντικές προσθήκες και βελτιώσεις στο πρότυπο MPI-1 καθώς και επεξηγήσεις σε τμήματα της προδιαγραφής MPI που είχαν υποστεί παρερμηνείες. [5]

1.3 Γενικές πτυχές του MPI

1.3.1 Στόχοι και πεδίο εφαρμογής του MPI

Οι κύριοι στόχοι του MPI είναι:

- Να προσφέρει φορητότητα στον πηγαίο κώδικα
- Να επιτρέψει την αποτελεσματική εφαρμογή σε μια σειρά αρχιτεκτονικών

Προσφέρει επίσης:

- Πολλή λειτουργικότητα
- Υποστήριξη για ετερογενείς παράλληλες αρχιτεκτονικές

1.3.2 Τι υπάρχει μέσα στο MPI

Το MPI ορίστηκε να περιλαμβάνει σχετικά μεγάλο αριθμό χαρακτηριστικών που αποδείχθηκαν χρήσιμα σε διάφορες υπάρχουσες βιβλιοθήκες μεταβίβασης μηνυμάτων. Σύμφωνα με αυτό, το πρότυπο MPI έχει περίπου 125 λειτουργίες. Για να το κρατήσουν διαχειρίσιμο, οι σχεδιαστές του MPI προσπάθησαν να κάνουν τα χαρακτηριστικά του σταθερά και ανεξάρτητα. Αυτό σημαίνει ότι οι χρήστες μπορούν να προσθέτουν σταδιακά ομάδες λειτουργιών στο ρεπερτόριό τους, ανάλογα με τις ανάγκες, χωρίς να μαθαίνουν τα πάντα εξαρχής. Η MPI ασχολείται με τους ακόλουθους τομείς:

- ***Μεταβίβαση μηνυμάτων από σημείο σε σημείο.*** Η απλούστερη μορφή μηνύματος είναι η επικοινωνία από σημείο σε σημείο. Ένα μήνυμα αποστέλλεται από μια διαδικασία αποστολής σε μια διαδικασία λήψης. Μόνο αυτές οι δύο διαδικασίες πρέπει να γνωρίζουν τα πάντα για το μήνυμα. Οι δύο βασικές λειτουργίες είναι η αποστολή και η λήψη, αλλά υπάρχουν μερικές διαφορετικές εκδοχές που αντιπροσωπεύουν διαφορετική σημασιολογία στην επικοινωνία.
- ***Συλλογική επικοινωνία.*** Μια αποδεδειγμένη ιδέα από τις υπάρχουσες βιβλιοθήκες μεταβίβασης μηνυμάτων είναι η έννοια της συλλογικής λειτουργίας, που εκτελείται από όλες τις διαδικασίες σε έναν υπολογισμό, δηλαδή επιτρέπει μεγαλύτερο αριθμό διεργασιών να επικοινωνήσουν (π.χ. broadcast λειτουργία). Οι συλλογικές λειτουργίες είναι δύο ειδών:
    - *Οι λειτουργίες μετακίνησης δεδομένων χρησιμοποιούνται για την αναδιάταξη των δεδομένων μεταξύ των διεργασιών.* Η απλούστερη από αυτές είναι η broadcast, αλλά μπορούν να καθοριστούν και να υποστηριχτούν από το MPI και άλλες λειτουργίες όπως το scattering και το gathering.
    - *Οι συλλογικές λειτουργίες χρησιμοποιούνται προκειμένου να υπολογιστεί μια τιμή από τα δεδομένα που εντοπίζονται σε διαφορετικές διεργασίες*, π.χ. το ελάχιστο, το μέγιστο, το άθροισμα, το λογικό OR, κλπ., καθώς και λειτουργίες που καθορίζονται από τον χρήστη.

Όλες αυτές οι λειτουργίες μπορούν να κατασκευαστούν από επικοινωνίες από σημείο σε σημείο, αλλά είναι μια καλή ιδέα να χρησιμοποιηθούν ρουτίνες αν υπάρχουν. (Για παράδειγμα, η βιβλιοθήκη που μεταβιβάζει μηνύματα μπορεί να εκμεταλλευτεί τις γνώσεις της σχετικά με τη δομή της μηχανής για να βελτιστοποιήσει και να αυξήσει τον παραλληλισμό σε αυτές τις λειτουργίες).

- ***Υποστήριξη ομάδων διεργασιών.*** Οι διεργασίες ανήκουν σε ομάδες. Μια ομάδα διεργασιών είναι μια διατεταγμένη συλλογή διαδικασιών και κάθε διαδικασία αναγνωρίζεται μοναδικά από την τάξη της (rank) μέσα στην κατάταξη. Για μια ομάδα n διεργασιών οι τάξεις κυμαίνονται από 0 έως n - 1. Οι ομάδες διεργασιών μπορούν να χρησιμοποιηθούν με δύο σημαντικούς τρόπους. Πρώτον, μπορούν να χρησιμοποιηθούν για να καθορίσουν ποιες διεργασίες εμπλέκονται σε μια συλλογική επικοινωνία, όπως η broadcast. Δεύτερον, μπορούν να εισαγάγουν παράλληλο έργο σε μια εφαρμογή, ώστε διαφορετικές ομάδες να εκτελούν διαφορετικές εργασίες.
- ***Υποστήριξη για περιβάλλοντα επικοινωνίας.*** Τα πλαίσια επικοινωνίας χρησιμοποιούνται για να διαχωριστούν οι οικογένειες των μηνυμάτων. Προωθούν τη λειτουργικότητα του λογισμικού επιτρέποντας την κατασκευή ανεξάρτητων ροών επικοινωνίας μεταξύ των διεργασιών, εξασφαλίζοντας έτσι ότι τα μηνύματα που αποστέλλονται σε μία φάση μιας εφαρμογής δεν παρεμποδίζονται εσφαλμένα από μια άλλη φάση. Επίσης δίνουν τη δυνατότητα στους συγγραφείς βιβλιοθηκών να γράψουν για πρώτη φορά παράλληλες βιβλιοθήκες που είναι εντελώς αυτόνομες από τον κώδικα του χρήστη και διαλειτουργικές με άλλες βιβλιοθήκες.
- ***Υποστήριξη για τοπολογίες εφαρμογών.*** Σε πολλές εφαρμογές οι διεργασίες διαμορφώνονται έχοντας μια συγκεκριμένη τοπολογία, όπως για παράδειγμα ένα δισδιάστατο ή τρισδιάστατο πλέγμα. Το MPI παρέχει υποστήριξη για γενικές τοπολογίες εφαρμογής που καθορίζονται από ένα γράφημα στο οποίο οι διεργασίες που επικοινωνούν συνδέονται με ένα τόξο. Οι τοπολογίες παρέχουν μια μέθοδο υψηλού επιπέδου για τη διαχείριση ομάδων διεργασιών χωρίς άμεση αντιμετώπισή τους.
- ***Προφίλ διεπαφής.*** Το MPI φόρουμ διαπίστωσε ότι ο προσδιορισμός προφίλ και άλλων μορφών μέτρησης της απόδοσης ήταν ιδιαίτερης σημασίας για την επιτυχία του MPI. Ταυτόχρονα, έμοιαζε ότι ήταν πολύ νωρίς για την τυποποίηση σε οποιαδήποτε προσέγγιση μέτρησης της απόδοσης. Κοινή σε όλες τις προσεγγίσεις, ωστόσο, είναι η απαίτηση ότι κάτι ιδιαίτερο συμβαίνει κατά τη στιγμή κάθε κλήσης MPI σε μια εφαρμογή. Ως εκ τούτου, το MPI Forum αποφάσισε να συμπεριλάβει στο

MPI προδιαγραφές για το πώς θα ήταν δυνατό για οποιονδήποτε να παρακολουθεί τις κλήσεις προς τη βιβλιοθήκη MPI και να εκτελεί αυθαίρετες ενέργειες.

1.3.3 Τι δεν υπάρχει στο MPI;

Έμμεσα εκτός του πεδίου εφαρμογής του MPI υπάρχει οποιαδήποτε ρητή υποστήριξη για:

- Αρχική φόρτωση των διεργασιών στους επεξεργαστές
- Αναπαραγωγή των διεργασιών κατά την εκτέλεση
- Multithreading (αλλά το MPI έχει σχεδιαστεί για να είναι ασφαλές για τα νήματα)
- Παράλληλα I / O
- Ενεργά μηνύματα
- Εικονική κοινόχρηστη μνήμη

1.3.4 Μοντέλο διεργασίας και ομάδες

Στο MPI η θεμελιώδης υπολογιστική μονάδα είναι η διεργασία. Όταν λέμε διεργασία εννοούμε τη μικρότερη διευθυνσιοδοτημένη μονάδα υπολογισμού. Κάθε διεργασία έχει ένα αυτόνομο νήμα ελέγχου και έναν ευδιάκριτο χώρο διεύθυνσης. Το μοντέλο διεργασίας MPI είναι στατικό, οπότε όσον αφορά την σχετική εφαρμογή, υπάρχει ένας σταθερός αριθμός διεργασιών από την έναρξη του προγράμματος έως την ολοκλήρωση. Κάθε διεργασία μπορεί να εκτελέσει τον δικό της ξεχωριστό κώδικα σε στυλ MIMD, ωστόσο το MPI δεν παρέχει μηχανισμούς για τη φόρτωση εκτελέσιμου κώδικα στους επεξεργαστές ούτε αναθέτει διεργασίες σε επεξεργαστές.

Οι διεργασίες ανήκουν σε ομάδες και μάλιστα κάθε διεργασία μπορεί να ανήκει σε πολλές ομάδες ταυτόχρονα. Αν και το μοντέλο διεργασίας MPI είναι στατικό, οι ομάδες διεργασιών είναι δυναμικές υπό την έννοια ότι μπορούν να δημιουργηθούν και να καταστραφούν. Ωστόσο, η ιδιότητα μέλους μιας ομάδας είναι στατική: για μία ή περισσότερες διεργασίες για να συμμετέχετε ή να αποχωρήσετε από μια ομάδα, πρέπει να οριστεί μια νέα ομάδα αντί να τροποποιήσετε την αρχική.

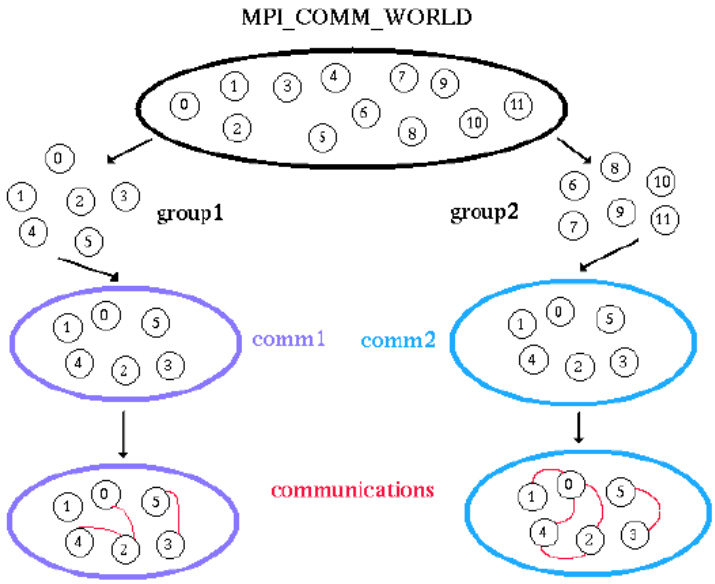
Οι διεργασίες προσδιορίζονται από δύο ιδιότητες: μία ομάδα και έναν βαθμό διεργασίας(rank) που είναι σχετικά με την ομάδα. Για μια ομάδα n διεργασιών οι βαθμοί κυμαίνονται από το 0 έως το n - 1.

1.3.5 Διαχωρισμός των οικογενειών των μηνυμάτων

Σχεδόν όλα τα συστήματα μεταβίβασης μηνυμάτων παρέχουν ένα όρισμα ετικέτας(tag) για τις λειτουργίες αποστολής και λήψης. Αυτό το όρισμα επιτρέπει στον προγραμματιστή να ασχοληθεί με την άφιξη των μηνυμάτων κατά τρόπο ομαλό, ακόμα και αν η άφιξη των μηνυμάτων δεν είναι με τη σειρά που επιθυμούμε. Το σύστημα μεταβίβασης μηνυμάτων παραβλέπει τα μηνύματα που φτάνουν "από λάθος ετικέτα μηνύματος" έως ότου το πρόγραμμα να είναι έτοιμο. Το MPI επεκτείνει την έννοια της ετικέτας με μια νέα έννοια: το πλαίσιο. Τα πλαίσια κατανέμονται κατά το χρόνο εκτέλεσης από το σύστημα ως απόκριση στα αιτήματα των χρηστών και χρησιμοποιούνται για την αντιστοίχιση μηνυμάτων. Ξεχωρίζουν το "χώρο ετικετών των μηνυμάτων" επιτρέποντας τη δημιουργία αυτόνομων χώρων ετικετών. Στο MPI, μια ετικέτα μηνύματος καθορίζεται από το πλαίσιο του μηνύματος και από την ετικέτα που είναι σχετική με το περιβάλλον.

1.3.6 Πεδίο επικοινωνίας

Το "πεδίο" μιας λειτουργίας επικοινωνίας προσδιορίζεται από το χρησιμοποιούμενο πλαίσιο και την ομάδα διεργασίας που εμπλέκεται. Η έννοια του πλαισίου και της ομάδας συνδυάζονται σε ένα ενιαίο αντικείμενο που ονομάζεται **επικοινωνητής(Communicator)**, το οποίο γίνεται ένα όρισμα στις περισσότερες λειτουργίες από σημείο σε σημείο καθώς και στις συλλογικές. Τα πλαίσια δεν είναι ορατά σε επίπεδο εφαρμογής, αλλά είναι πάντα κρυμμένα στα αντικείμενα επικοινωνίας. Έτσι ο προγραμματιστής πρέπει να ασχοληθεί με τους επικοινωνητές αντί με τα πλαίσια. Όταν ένα νέο αντικείμενο επικοινωνητή κατασκευάζεται από τον προγραμματιστή, το σύστημα δημιουργεί ένα μοναδικό πλαίσιο και το βάζει αυτόματα στον επικοινωνητή. Ο communicator που ορίζεται βασικά σε κάθε



Σχήμα 4: Ο επικοινωνητής MPI_COMM_WORLD[10]

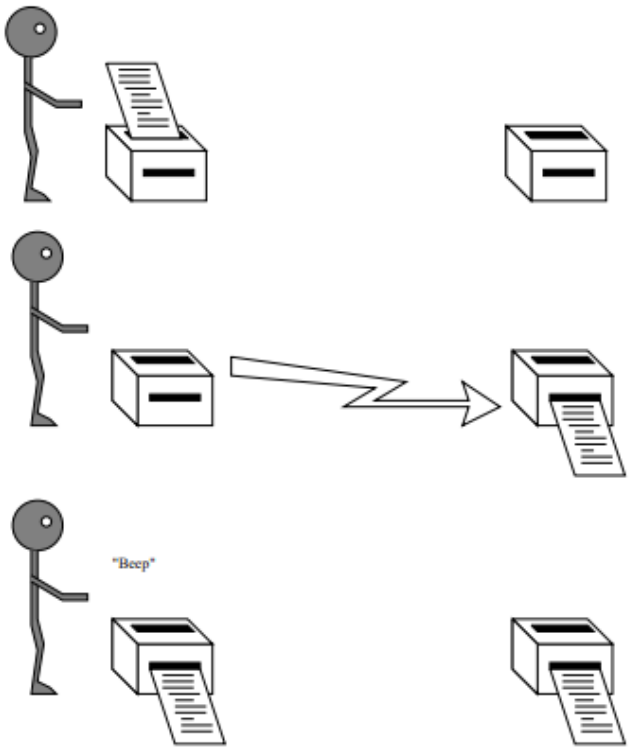
πρόγραμμα MPI είναι ο MPI_COMM_WORLD, ο οποίος περιλαμβάνει όλες τις διεργασίες που συμμετέχουν στην εκτέλεση του προγράμματος. Ο MPI_COMM_WORLD φαίνεται στο παραπάνω σχήμα.[9]

1.4 Επικοινωνία στο MPI

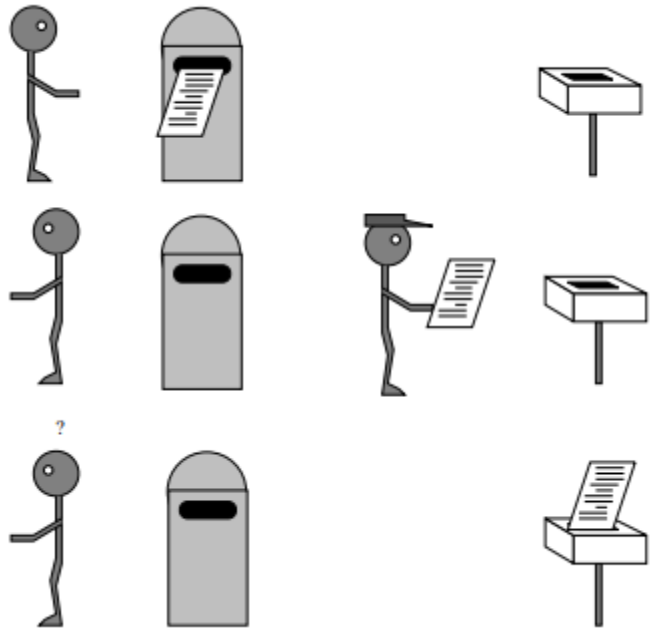
1.4.1 Επικοινωνία από σημείο σε σημείο (point to point communication)

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** και αποτελεί το μοντέλο επικοινωνίας για ένα πολύ μεγάλο εύρος εφαρμογών MPI. Υπάρχουν αρκετοί τρόποι επικοινωνίας από σημείο σε σημείο.

Η πιο σημαντική διάκριση στους τρόπους επικοινωνίας είναι η **σύγχρονη** και η **ασύγχρονη**. Στη σύγχρονη επικοινωνία, περιλαμβάνονται πληροφορίες σχετικές με την ολοκλήρωση της αποστολής του μηνύματος. Στην ασύγχρονη επικοινωνία, είναι γνωστό μόνο το πότε έφυγε το μήνυμα, όχι το αν έφτασε.



Σχήμα 5: Μια σύγχρονη επικοινωνία δεν ολοκληρώνεται μέχρι να ληφθεί το μήνυμα.[7]

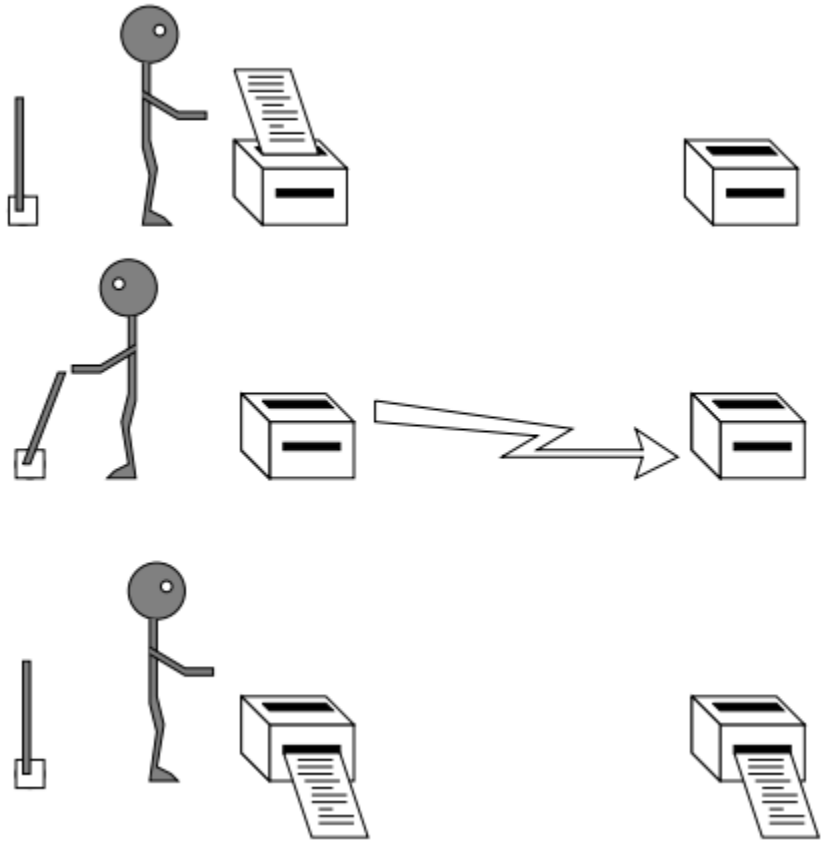


Σχήμα 6: Μια ασύγχρονη επικοινωνία ολοκληρώνεται όταν το μήνυμα βρίσκεται σε εξέλιξη[7]

Άλλη μια σημαντική διάκριση στους τρόπους επικοινωνίας είναι το κλείδωμα ή όχι της επικοινωνίας. Στην περίπτωση των διεργασιών από σημείο σε σημείο, οι συναρτήσεις του MPI εμφανίζονται σε δύο διαφορετικές μορφές που φέρουν τα ονόματα **παρεμποδιστικές** (blocking) και **μη παρεμποδιστικές** (non blocking) συναρτήσεις. 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. Στην αντίθετη περίπτωση, κατά την οποία η κλήση της συνάρτησης δεν προκαλεί την αναστολή της λειτουργίας της διεργασίας, η εν λόγω συνάρτηση λέγεται μη παρεμποδιστική. Εντελώς ανάλογη με την προηγούμενη κατάσταση, είναι και αυτή που συσχετίζεται με τον τρόπο λειτουργίας της διεργασίας παραλήπτη. Εάν η εν λόγω διεργασία χρησιμοποιεί την παρεμποδιστική έκδοση της receive – η οποία νοείται ως η συνάρτηση που διαβάζει το περιεχόμενο του μηνύματος που έχει σταλεί – τότε η λειτουργία της διεργασίας παραλήπτη θα ανασταλεί κατά το χρονικό διάστημα ανάγνωσης της πληροφορίας. Όταν η διακινούμενη πληροφορία έχει παραληφθεί στο σύνολό της από τη διεργασία παραλήπτη, η συνάρτηση receive θα αναστείλει τη λειτουργία της κάτι που θα επιτρέψει τη συνέχιση και τελικά την ολοκλήρωση της λειτουργίας της διεργασίας παραλήπτη.

Στην περίπτωση κατά την οποία χρησιμοποιούνται οι μη παρεμποδιστικές εκδόσεις των συναρτήσεων αποστολής και παραλαβής μηνυμάτων, η κατάσταση είναι εντελώς

διαφορετική. Στην περίπτωση αυτή η κλήση της μη παρεμποδιστικής έκδοσης της send, δεν αναστέλλει τη λειτουργία της διεργασίας αποστολέα, η οποία συνεχίζεται κανονικά και εκτελείται παράλληλα με την αποστολή του μηνύματος.



Σχήμα 7: Η επικοινωνία μη μπλοκαρίσματος επιτρέπει τη διεκπεραίωση χρήσιμων εργασιών κατά την αναμονή της ολοκλήρωσης της επικοινωνίας[7]

Οι κανονικές συσκευές φαξ για παράδειγμα παρέχουν επικοινωνία μπλοκαρίσματος. Το φαξ παραμένει απασχολημένο μέχρι να σταλεί το μήνυμα. Ορισμένες σύγχρονες συσκευές φαξ περιέχουν μνήμη. Αυτό επιτρέπει να φορτωθεί ένα έγγραφο στη μνήμη και εάν ο απομακρυσμένος αριθμός είναι ενεργοποιημένος, το μηχάνημα μπορεί να παραμείνει για να συνεχίσει να προσπαθεί μέχρι να τελειώσει έτσι ώστε κι εμείς να κάνουμε κάτι πιο σημαντικό. Πρόκειται για μια μη-παρεμποδιστική λειτουργία.

Η λήψη ενός μηνύματος μπορεί επίσης να είναι μια μη-παρεμποδιστική λειτουργία. Για παράδειγμα, ενεργοποιώντας τη συσκευή φαξ και αφήνοντας τη ανοιχτή, ώστε να μπορεί να φτάσει το μήνυμα. Στη συνέχεια, δοκιμάζετε περιοδικά τη συσκευή ώστε να διαπιστώσετε εάν έχει φτάσει το μήνυμα.[7]

Το MPI παρέχει ένα σύνολο συναρτήσεων που χρησιμοποιείται για την αποστολή και λήψη δεδομένων με τη μορφή μηνυμάτων. Κάθε ένα από αυτά τα μηνύματα έχει μία συγκεκριμένη μορφή καθώς διαθέτει μία ετικέτα (tag) που το διαφοροποιεί από τα υπόλοιπα μηνύματα της εφαρμογής. Αυτή η ετικέτα είναι πολύ σημαντική και ιδιαίτερα χρήσιμη σε περιπτώσεις επικοινωνίας διεργασιών που ανήκουν σε συστήματα που διαθέτουν διαφορετική αρχιτεκτονική και απαιτείται η πρότερη μετατροπή των μεταφερόμενων δεδομένων. Επίσης η ετικέτα κρίνεται σημαντική σε περιπτώσεις όπου απαιτείται η ταξινόμηση μηνυμάτων που παραλαμβάνονται με σειρά διαφορετική από εκείνη με την οποία έχουν σταλεί.

Μεταξύ των δεδομένων προς αποστολή που περιλαμβάνονται σε ένα μήνυμα του προτύπου MPI, περιέχονται και άλλες πληροφορίες που είναι αναγκαίες για την επιτυχή μετάδοση του μηνύματος. Οι πληροφορίες αυτές περιλαμβάνουν το πλήθος και τον τύπο των δεδομένων που περιέχονται στο μήνυμα, το βαθμό των διεργασιών αποστολής και λήψης, την ετικέτα του μηνύματος και το όνομα του επικοινωνητή που χρησιμοποιείται μέσω του οποίου επικοινωνούν οι δύο διεργασίες. Οι πληροφορίες αυτές μεταφέρονται ως ορίσματα στη συνάρτηση αποστολής του μηνύματος, ενώ ανάλογη είναι και η συνάρτηση παραλαβής του μηνύματος, η οποία ωστόσο περιέχει κι ένα επιπλέον όρισμα που περιλαμβάνει πληροφορίες σχετικές με τη διαδικασία παραλαβής. Χρησιμοποιώντας αυτές τις δύο συναρτήσεις καθώς και κάποιες άλλες που χρησιμοποιούνται για την αρχικοποίηση και τον τερματισμό του προτύπου MPI, μπορούμε να δημιουργήσουμε όλες σχεδόν τις εφαρμογές που σχετίζονται με αυτή τη διαδικασία ανταλλαγής μηνυμάτων.[3]

1.4.2 Συλλογικές επικοινωνίες (collective communications)

Το δεύτερο είδος επικοινωνίας που υποστηρίζει το πρωτόκολλο MPI, είναι οι **συλλογικές επικοινωνίες** οι οποίες περιλαμβάνουν περισσότερες από δύο διεργασίες. Οι πιο βασικές από αυτές τις μορφές επικοινωνίας, είναι η **εκπομπή (broadcasting)**, όπου ένα μήνυμα αποστέλλεται ενιαίο σε όλες τις διεργασίες, η **διασπορά (scattering)** όπου ένα μήνυμα χωρίζεται σε μικρότερα κομμάτια όπου το κάθε ένα αποστέλλεται και σε μια διαφορετική διεργασία, η **συλλογή (gather)** όπου μία απλή διεργασία συγκεντρώνει όλα τα μηνύματα που αποστέλλονται από όλα τα μέλη μιας ομάδας διεργασιών, και η **μείωση (reduce)** όπου μια διεργασία συλλέγει δεδομένα από τις υπόλοιπες διεργασίες της ομάδας

και ταυτόχρονα υπολογίζει κάποια συνάρτηση αυτών των δεδομένων (για παράδειγμα την εύρεση μίας μέγιστης τιμής). 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. [3]

1.5 Έναρξη γνωριμίας με το MPI

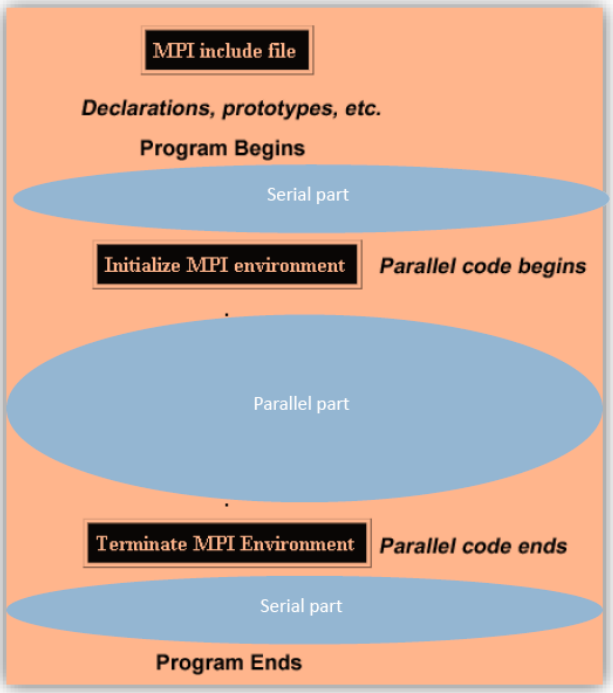
1.5.1 Εγκατάσταση πακέτου

Καταρχήν θα πρέπει να κατεβάσουμε την έκδοση mpich-3.2 από το https://www.mpich.org/downloads/ και να εκτελέσουμε την εντολή $sudo make install προκειμένου να εγκαταστήσουμε το πρόγραμμα στον υπολογιστή μας. Για να επιβεβαιώσουμε ότι εγκαταστάθηκε σωστά εκτελούμε την εντολή mpiexec –version. [23]

1.5.2 Απλό πρόγραμμα στη γλώσσα C

Συνήθως όλες οι γλώσσες προγραμματισμού ξεκινάνε με μία παραλλαγή του γνωστού «Hello World»! Ως αποτέλεσμα της εκτέλεσης του απλού αυτού προγράμματος είναι η εκτύπωση στην οθόνη του υπολογιστή μας του μηνύματος Hello World. Στη γλώσσα C για παράδειγμα θα γράφαμε το παρακάτω:

```
1  #include <stdio.h>
2  int main(void){
3  printf("Hello World !! \n");
4  return 0;
5  }
```



Σχήμα 8: Η γενική δομή ενός προγράμματος MPI[3]

1.5.3 Η δομή ενός προγράμματος MPI

Η γενική δομή ενός προγράμματος MPI παρουσιάζεται στο παραπάνω σχήμα. Πιο συγκεκριμένα, μπορούμε να δούμε ότι ένα πρόγραμμα γραμμένο σε MPI περιλαμβάνει δύο κύρια μέρη. Το 1ο μέρος είναι το «σειριακό» και εκτελείται μόνο σε έναν επεξεργαστή, ενώ το 2ο μέρος είναι το «παράλληλο» και εκτελείται σε κάθε κόμβο ξεχωριστά.

Πιο συγκεκριμένα η δομή ενός προγράμματος που χρησιμοποιεί συναρτήσεις MPI είναι η ακόλουθη:

```
1  #include <mpi.h>
2  // Στο σημείο αυτό μπορούμε να δηλώσουμε τα επιθυμητά σε κάθε περίπτωση
3  // αρχεία επικεφαλίδας
4  int main (int argc, char * argv[]) {
5  // Δήλωση των μεταβλητών του προγράμματος
6  // Κλήση συναρτήσεων της γλώσσας C
7  MPI_Init (&argc, &argv);
8  //Ανάμεσα στις συναρτήσεις MPI_Init() και MPI_Finalize() μπορούμε να
9  //καλέσουμε συναρτήσεις MPI και φυσικά οποιαδήποτε συνάρτηση της γλώσσας C
10 MPI_Finalize();
11 // Στο σημείο αυτό μπορούμε να καλέσουμε συναρτήσεις της γλώσσας C αλλά όχι
12 // και συναρτήσεις του MPI
13 return (0);
14 }
```

Σε γλώσσα MPI ο πηγαίος κώδικας για το απλό αυτό πρόγραμμα που περιγράψαμε παραπάνω, παρουσιάζεται στη συνέχεια. Το αρχείο αποθηκεύεται με το όνομα hello.c.

```
1  #include <mpi.h>
2  #include <stdio.h>
3  int main (int argc, char * argv[0]) {
```

```
4  char message [16] = "Hello World !! "
5  MPI_Init (&argc, &argv);
6  printf ("%s\n", message);
7  MPI_Finalize();
8  return (0);
9  }
```

Στο παραπάνω πρόγραμμα παρατηρούμε ότι όντως έχουμε ένα πρόγραμμα C μιας και περιλαμβάνεται (include) το αρχείο κεφαλίδων stdio.h της καθιερωμένης C. Στην 1η γραμμή όμως υπάρχει και κάτι νέο όπως το αρχείο κεφαλίδων mpi.h. Αυτό το αρχείο περιλαμβάνει δηλώσεις των συναρτήσεων MPI, ορισμούς μακροεντολών και άλλα. Περιλαμβάνει όλες τις δηλώσεις και τους ορισμούς που απαιτούνται για τη μεταγλώττιση ενός προγράμματος MPI.

Κάτι ακόμα που παρατηρούμε είναι πως όλα τα αναγνωριστικά που ορίζονται από την MPI αρχίζουν με τη συμβολοσειρά MPI_. Το πρώτο γράμμα μετά τον χαρακτήρα της κάτω παύλας είναι κεφαλαίο όταν αφορά όνομα συνάρτησης ή τύπο της MPI. Τα ονόματα μακροεντολών και σταθερών της MPI είναι γραμμένα με όλα τα γράμματα κεφαλαία, επομένως είναι εμφανές ποια στοιχεία ορίζονται από την MPI και ποια από το πρόγραμμα χρήστη. [1]

1.5.4 Οι βασικές συναρτήσεις του MPI

Σύμφωνα με το [1], το MPI απαρτίζεται από ένα μεγάλο πλήθος συναρτήσεων, οι οποίες μπορούν να χρησιμοποιηθούν στη συγγραφή εφαρμογών. Παρόλα αυτά, αρκετά προγράμματα μπορούν να γραφούν με τη χρήση μόνο 6 βασικών συναρτήσεων.

Όνομα συνάρτησης	Σκοπός
MPI_INIT()	Αρχικοποίηση του MPI
MPI_Comm_rank()	Αριθμός διεργασίας
MPI_Comm_size()	Σύνολο διεργασιών
MPI_Recv()	Παραλαβή μηνυμάτων
MPI_Send()	Αποστολή μηνυμάτων
MPI_Finalize()	Τερματισμός του MPI

Πίνακας 1: Οι 6 βασικές συναρτήσεις του MPI[6]

Στο παραπάνω πρόγραμμα παρατηρούμε ότι οι μόνες δύο συναρτήσεις του MPI που χρησιμοποιούνται είναι η MPI_Init και η MPI_Finalize οι οποίες αρχικοποιούν και τερματίζουν απλά το προγραμματιστικό αυτό περιβάλλον αντίστοιχα.

Η **κλήση της MPI_Init** αρχικοποιεί το πρότυπο MPI και δημιουργεί όλες τις δομές δεδομένων που είναι αναγκαίες για τη λειτουργία του (όπως είναι για παράδειγμα ο επικοινωνητής MPI_COMM_WORLD). Ένας γενικός κανόνας είναι πως το πρόγραμμα δεν μπορεί να καλέσει καμία άλλη συνάρτηση MPI πριν καλέσει την MPI_Init. Μάλιστα η κλήση της γίνεται μία και μοναδική φορά.[2] Η σύνταξή της είναι:

```
int MPI_Init(int* argc_p , char***  argv_p)
```

Τα ορίσματα argc_p και argv_p είναι δείκτες προς τα ορίσματα argc και argv της main. Αν το πρόγραμμά μας όμως δε χρησιμοποιεί αυτά τα ορίσματα, τότε μπορούμε να μεταβιβάσουμε την τιμή NULL και για τα δύο. Η MPI_Init, όπως οι περισσότερες συναρτήσεις MPI, επιστρέφει έναν ακέραιο κωδικό σφάλματος, αλλά τις περισσότερες φορές παραλείπουμε αυτούς τους κωδικούς σφαλμάτων.

Από την άλλη η **κλήση προς την MPI_Finalize** ενημερώνει τον τερματισμό της λειτουργίας του προτύπου MPI. Αυτό σημαίνει ότι όποιοι πόροι έχουν δεσμευτεί για την MPI μπορούν να αποδεσμευτούν. Η σύνταξή της είναι πολύ απλή:

```
int MPI_Finalize(void);
```

Γενικά δεν πρέπει να καλούμε καμία άλλη συνάρτηση μετά την κλήση της MPI_Finalize. Επίσης, δεν είναι απαραίτητο οι κλήσεις των MPI_Init και MPI_Finalize να βρίσκονται στη main.

Προκειμένου να δημιουργήσουμε το εκτελέσιμο αρχείο, μπορούμε να χρησιμοποιήσουμε έναν οποιοδήποτε μεταγλωττιστή της γλώσσας C, όπως είναι για παράδειγμα ο μεταγλωττιστής gcc (GNU C compiler). Παρόλα αυτά το MPI περιέχει επιπρόσθετα κι έναν ενσωματωμένο C compiler που μαζί με τις λειτουργίες ενός τυπικού μεταγλωττιστή περιλαμβάνει κι εξειδικευμένες λειτουργίες που σχετίζονται με το περιβάλλον του προτύπου MPI. Ο μεταγλωττιστής αυτός ονομάζεται mpicc και χρησιμοποιείται ακριβώς όπως και ο μεταγλωττιστής gcc.

Έτσι αν για παράδειγμα θέλουμε να κάνουμε compile το hello.c και να δημιουργήσουμε ένα εκτελέσιμο αρχείο C με όνομα hello θα γράψουμε:

```
$ mpicc -o hello hello.c
```

Αν στη συνέχεια θελήσουμε να εκτελέσουμε το πρόγραμμα, μπορούμε να κάνουμε χρήση μία από τις εντολές **mpirun** και **mpiexec**. Τις πιο πολλές φορές το αποτέλεσμα που προκύπτει και με τις δύο εντολές είναι το ίδιο. Η διαφορά εναπόκειται στο ότι η mpiexec

ορίστηκε εξ αρχής στο πρότυπο MPI ενώ η mpirun προστέθηκε αργότερα σε διάφορες εκδοχές του MPI (Mpich, OpenMPI κλπ). Συνήθως όμως επειδή δεν είμαστε σίγουροι για το πώς θα συμπεριφερθεί η εντολή προτιμούμε την εντολή mpiexec.

Η σύνταξη της εντολής είναι η ακόλουθη:

$ mpiexec [**options**] <**program**> [<**args**>]

Το **program** είναι το όνομα του εκτελέσιμου αρχείου, το **args** είναι οι παράμετροι που αφορούν το εκτελέσιμο και είναι προαιρετικοί, ενώ το **options** είναι παράμετροι που σχετίζονται με την εντολή mpiexec και δέχονται συγκεκριμένες τιμές. Έτσι χρησιμοποιώντας το -np μπορούμε να ορίσουμε τον αριθμό των επεξεργαστών που θέλουμε. Προφανώς η τιμή πρέπει να είναι θετικός ακέραιος. Για παράδειγμα, αν θέλουμε να τρέξουμε το αρχείο με όνομα hello.c και χρειαζόμαστε 1 επεξεργαστή, γράφουμε [4]:

$ mpiexec -np 1 ./hello

ενώ για να εκτελέσουμε το πρόγραμμα με τέσσερις διεργασίες θα πληκτρολογήσουμε

$ mpiexec -np 4 ./hello

Αποτέλεσμα της εκτέλεσης της παραπάνω εντολής, είναι η εκτύπωση του μηνύματος Hello World !! από κάθε μια από αυτές τις διεργασίες. Συγκεκριμένα, με μία διεργασία, η έξοδος του προγράμματος θα ήταν:

Hello World !!

και με τέσσερις διεργασίες, η έξοδος θα ήταν

Hello World !!

Hello World !!

Hello World !!

Hello World !!

Δύο άλλες ενδιαφέρουσες και πολύ χρήσιμες συναρτήσεις του MPI είναι η **MPI_Comm_rank** και η **MPI_Comm_size**. Η σύνταξή τους είναι:

```
int MPI_Comm_size(MPI_Comm comm, int* comm_sz_p);

int MPI_Comm_rank(MPI_Comm comm, int* my_rank_p);
```

Οι συναρτήσεις αυτές δέχονται δύο ορίσματα όπου το πρώτο όρισμα είναι ένας επικοινωνητής ειδικού τύπου που ορίζει η MPI, του MPI_Comm. Το δεύτερο όρισμα

διαφέρει στις δύο αυτές συναρτήσεις. Στη μεν MPI_Comm_size το δεύτερο όρισμα επιστρέφει το πλήθος των διεργασιών του επικοινωνητή, στη δε MPI_Comm_rank το δεύτερο όρισμα επιστρέφει την τάξη της εκάστοτε κληθείσης διεργασίας στο σύνολο διεργασιών του επικοινωνητή.

Εάν για παράδειγμα δημιουργήσουμε πέντε αντίγραφα της διεργασίας hello χρησιμοποιώντας την εντολή $mpiexec -np 5 ./hello, η κλήση της συνάρτησης MPI_Comm_size θα αποδώσει στην ακέραια μεταβλητή size την τιμή 5, ενώ η κλήση της συνάρτησης MPI_Comm_rank θα αποδώσει στη μεταβλητή rank την τάξη της τρέχουσας διεργασίας που είναι διαφορετική για κάθε διεργασία. Δηλαδή η μεταβλητή rank της πρώτης διεργασίας θα πάρει την τιμή 0, η μεταβλητή rank της δεύτερης διεργασίας θα πάρει την τιμή 1, η μεταβλητή rank της τρίτης διεργασίας θα πάρει την τιμή 2, η μεταβλητή rank της τέταρτης διεργασίας θα πάρει την τιμή 3, και η μεταβλητή rank της πέμπτης διεργασίας θα πάρει την τιμή 4.

Στη συνέχεια για να κατανοήσουμε καλύτερα το πώς χρησιμοποιούνται οι συναρτήσεις MPI_Comm_rank και MPI_Comm_size, θα επεκτείνουμε το παραπάνω πρόγραμμα ώστε να εμφανίζει επιπλέον την τάξη της τρέχουσας διεργασίας και το συνολικό πλήθος των διεργασιών του Communicator. Θα ορίσουμε λοιπόν δύο ακέραιες μεταβλητές rank και size και θα τις περάσουμε ως ορίσματα στις αντίστοιχες συναρτήσεις. Ο νέος κώδικας που θα προκύψει μετά τις παραπάνω προσθήκες θα έχει την παρακάτω μορφή:

```
1   #include <mpi.h>
2   #include <stdio.h>
3   int main (int argc, char * argv[0]) {
4   int rank, size;
5   MPI_Init (&argc, &argv);
6   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
7   MPI_Comm_size (MPI_COMM_WORLD, &size);
8   printf (“Hello World from process %d of %d\n”, rank, size);
8   MPI_Finalize();
10  return (0);
11  }
```

Εάν λοιπόν μεταγλωττίσουμε το παραπάνω αρχείο και το τρέξουμε χρησιμοποιώντας την εντολή $ mpiexec -np 5 ./hello, το αποτέλεσμα που θα λάβουμε θα έχει τη μορφή

Hello World from process 1 of 5
Hello World from process 2 of 5
Hello World from process 3 of 5
Hello World from process 4 of 5
Hello World from process 5 of 5

Αυτό που παρατηρούμε κατά την εκτύπωση είναι πως η κάθε διεργασία εκτός από το μήνυμα Hello World έχει το δικό της βαθμό καθώς και το συνολικό πλήθος των διεργασιών του Communicator.

Το παράδειγμα αυτό χρησιμοποιεί επίσης την τεχνική **SPMD: Single Program – Multiple Data**. Με την τεχνική αυτή, είναι δυνατή η εκτέλεση διαφορετικού κώδικα σε διαφορετικούς επεξεργαστές με τη χρήση ενός και μοναδικού προγράμματος, χρησιμοποιώντας δομές ελέγχου ροής. Έτσι οι διεργασίες που εκτελούνται στους διαφόρους κόμβους, εκτελούν διαφορετικό κώδικα παρόλο που δεν εκτελούν διαφορετικά προγράμματα. [9]

Ολοκληρώνοντας θα αναφερθούμε σε δύο ακόμη βασικές συναρτήσεις που ανήκουν στην κατηγορία των παρεμποδιστικών συναρτήσεων αποστολής και λήψης μηνυμάτων, τις **MPI_Send** και **MPI_Recv** αντίστοιχα. Τη συνάρτηση αποστολής τη χρησιμοποιούμε προκείμενου να στείλουμε ένα μήνυμα σε μια από τις διεργασίες που περιέχονται στον τρέχοντα Communicator, ενώ τη συνάρτηση λήψης τη χρησιμοποιούμε προκείμενου να παραλάβουμε ένα μήνυμα από κάποια από τις διεργασίες που ανήκουν στον τρέχοντα Communicator. [2]

Στη συνέχεια θα διαφοροποιήσουμε το πρόγραμμα του «hello world» έτσι ώστε να χρησιμοποιεί την MPI. Όμως αντί να ρυθμίσουμε κάθε διεργασία να τυπώνει απλώς το μήνυμά της, θα αναθέσουμε την έξοδο σε μία μόνο διεργασία, στην οποία θα στέλνουν τα μηνύματά τους οι υπόλοιπες. Το ρόλο αυτής της διεργασίας θα τον έχει η διεργασία 0. Ας δούμε το σχετικό πρόγραμμα.[1]

```
1    #include <stdio.h>
2    #include <string.h>     /* Για τη strlen                    */
3    #include <mpi.h>        /* Για συναρτήσεις MPI, κλπ */
4
5    int main (int argc, char **argv) {
```

```
6        char message[100];
7        int  size;   /* Πλήθος διεργασιών                    */
8        int  rank;   /* Αριθμός τρέχουσας διεργασίας   */
9           MPI_Init (&argc, &argv);
10          MPI_Comm_size (MPI_COMM_WORLD, &size);
11          MPI_Comm_rank (MPI_COMM_WORLD, &rank);
12
13          if (rank !=0) {
14             sprintf(message, "Process %d of %d says hello.", rank, size);
15            MPI_Send(message, strlen(message) + 1, MPI_CHAR, 0, 0,
16            MPI_COMM_WORLD);
17          } else {
18                 printf("This is process %d waiting for messages\n", rank);
19                 for (int i = 1; i < size; i++) {
20                 MPI_Recv(message, 100, MPI_CHAR, i, 0, MPI_COMM_WORLD,
21                 MPI_STATUS_IGNORE);
22                 printf("%s\n", message);
23                 }
24           }
25          MPI_Finalize();
26          return 0;
27       }   /* main */
```

Πιο συγκεκριμένα, κάθε αποστολή πραγματοποιείται μέσω μιας κλήσης στην **MPI_Send**, της οποίας η σύνταξη είναι

int MPI_Send(void* *msg_buffer*, int *msg_size*, MPI_Datatype *msg_type*, int *dest*, int *tag*, MPI_Comm *communicator*);

Τα τρία πρώτα ορίσματα, τα msg_buffer, msg_size και msg_type, καθορίζουν το περιεχόμενο του μηνύματος. Τα υπόλοιπα ορίσματα, τα dest, tag και communicator, καθορίζουν τον παραλήπτη του μηνύματος.

Το πρώτο όρισμα, το *msg_buffer*, καθορίζει εκείνη την περιοχή μνήμης που βρίσκονται τα περιεχόμενα του μηνύματος. Στο πρόγραμμά μας, είναι απλώς η

συμβολοσειρά message που περιέχει το μήνυμα. Το δεύτερο και το τρίτο όρισμα, δηλαδή τα msg_size και msg_type, προσδιορίζουν την ποσότητα των δεδομένων που θα σταλούν. Στο πρόγραμμά μας, το όρισμα *msg_size* είναι το πλήθος των χαρακτήρων του μηνύματος συν έναν, τον χαρακτήρα '\0' με τον οποίο τερματίζονται οι συμβολοσειρές της C. Το όρισμα *msg_type* έχει τιμή τον τύπο MPI_CHAR. Ο συνδυασμός αυτών των δύο ορισμάτων ενημερώνει το σύστημα ότι το μήνυμα περιέχει strlen (message) +1 χαρακτήρες.

Εφόσον οι τύποι της C(int, char, κ.ο.κ) δεν μπορούν να μεταβιβάζονται ως ορίσματα συναρτήσεων, η MPI ορίζει έναν ειδικό τύπο, τον MPI_Datatype, ο οποίος χρησιμοποιείται για το όρισμα msg_type. Η MPI ορίζει επίσης και μερικές σταθερές τιμές γι' αυτόν τον τύπο. Μερικές από αυτές φαίνονται στον παρακάτω πίνακα:

Τύπος δεδομένων MPI	**Τύπος δεδομένων C**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long 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
MPI_BYTE	
MPI_PACKED	

Πίνακας 2: Μερικοί προκαθορισμένοι τύποι δεδομένων της MPI[1]

Το πλεονέκτημα της χρήσης των ειδικών τύπων του MPI αντί των τύπων της C είναι ότι το MPI μπορεί έτσι να υποστηρίξει διαφορετικού τύπου πλατφόρμες στις οποίες πιθανώς οι τύποι της C να αναπαριστώνται με διαφορετικό τρόπο.

Το τέταρτο όρισμα, το *dest*, προσδιορίζει το βαθμό της διεργασίας για την οποία προορίζεται το μήνυμα. Το πέμπτο όρισμα, το *tag* (ετικέτα), είναι ένας μη αρνητικός ακέραιος. Δίνει τη δυνατότητα να τροποποιούμε μηνύματα που έχουν πανομοιότυπο περιεχόμενο.

Τέλος, το τελευταίο όρισμα της MPI_Send είναι ένας ***επικοινωνητής***, που υπάρχει σε όλες τις συναρτήσεις MPI που σχετίζονται με την επικοινωνία. Η επικοινωνία μεταξύ διεργασιών μπορεί να στεφθεί με επιτυχία μόνο αν χρησιμοποιούν ίδιο επικοινωνητή. Αυτή η δυνατότητα επιτρέπει σε πιο σύνθετα προγράμματα να βεβαιώνουν ότι τα μηνύματα δε θα λαμβάνονται τυχαία από λανθασμένους παραλήπτες.

Στη συνέχεια θα αναφερθούμε στη συνάρτηση **MPI_Recv**. Τα έξι πρώτα ορίσματα της MPI_Recv αντιστοιχούν στα πρώτα έξι ορίσματα της MPI_Send:

int MPI_Recv(**void*** *msg_buffer*, **int** *buf_size*, MPI_Datatype *buf_type*, **int** *source*, **int** *tag*, MPI_Comm *communicator*, MPI_Status* *status_p*);

Τα πρώτα τρία ορίσματα καθορίζουν τη διαθέσιμη μνήμη για τη λήψη του μηνύματος. Συγκεκριμένα, το *msg_buffer* δείχνει στην κατάλληλη περιοχή μνήμης, το *buf_size* προσδιορίζει το πλήθος των αντικειμένων που μπορούν να αποθηκευτούν σε αυτή την περιοχή και το *buf_type* προσδιορίζει τον τύπο αυτών των αντικειμένων. Τα επόμενα τρία ορίσματα λειτουργούν περισσότερο ως αναγνωριστικά του μηνύματος. Συγκεκριμένα, το όρισμα *source* καθορίζει τη διεργασία από όπου θα πάρουμε το μήνυμα, το όρισμα *tag* πρέπει να ταιριάζει με το αντίστοιχο όρισμα του μηνύματος που στέλνουμε, και το όρισμα *communicator* πρέπει να αντιστοιχεί στον επικοινωνητή που χρησιμοποιήθηκε από τη διεργασία-αποστολέα. Όσον αφορά το όρισμα *status_p* δε χρησιμοποιείται από την καλούσα συνάρτηση, και, όπως στο παράδειγμά μας με το πρόγραμμα “hello world”, στη θέση του μπορούμε να μεταβιβάζουμε τη σταθερά MPI_STATUS_IGNORE της MPI.

1.5.5 Αντιστοίχηση μηνυμάτων

Ας υποθέσουμε ότι η διεργασία A καλεί την MPI_Send ως εξής:

MPI_Send (send_buf, send_buf_size, send_type, dest, send_tag, send_comm);

Επίσης, έστω ότι η διεργασία B καλεί την MPI_Recv με ορίσματα

MPI_Recv (recv_buf, recv_buf_size, recv_type, src, recv _tag, recv_comm, &status);

Για να μπορέσει να λάβει η B το μήνυμα που έστειλε η A με τη χρήση των παραπάνω συναρτήσεων, θα πρέπει να ισχύουν οι συνθήκες:

recv_comm = send_comm,

recv_tag = send_tag,

dest = b, και

src = a

Πιο συγκεκριμένα θα πρέπει να ισχύει ο παρακάτω κανόνας:

recv_type = send_type και recv_buf_size>=send_buf_ size

Γενικά, προκειμένου να υπάρξει επιτυχής επικοινωνία θα πρέπει τα τρία πρώτα ζεύγη ορισμάτων και στις δύο συναρτήσεις να προσδιορίζουν συμβατές περιοχές προσωρινής αποθήκευσης

Βέβαια υπάρχει το ενδεχόμενο μια διεργασία να λαμβάνει μηνύματα από πολλές άλλες διεργασίες και να μη γνωρίζει τη σειρά με την οποία οι άλλες διεργασίες στέλνουν αυτά τα μηνύματα και να πρέπει να περιμένει μέχρι οι υπόλοιπες διεργασίες να τελειώσουν τις δουλειές τους. Για την αποφυγή αυτού του προβλήματος που μπορεί να προκαλέσει μεγάλη καθυστέρηση στο πρόγραμμά μας, η MPI παρέχει μια ειδική σταθερά, την **MPI_ANY_SOURCE**, την οποία μπορούμε να μεταβιβάζουμε στην MPI_Recv. Τότε, αν η διεργασία 0 εκτελεί τον παρακάτω κώδικα, θα μπορεί να λαμβάνει τα αποτελέσματα με τη σειρά με την οποία ολοκληρώνονται οι διεργασίες:

```
for ( i=1; i < comm_size; i++) {
    MPI_Recv (result, result_size, result_type, MPI_ANY_SOURCE, result_tag, comm,
    MPI_STATUS_IGNORE);
    process_result(result);
}
```

Παρόμοια, είναι δυνατό μια διεργασία να λαμβάνει πολλά μηνύματα με διαφορετικές ετικέτες από μια άλλη διεργασία και να μη γνωρίζει τη σειρά αποστολής αυτών των μηνυμάτων. Γι’ αυτή την περίπτωση, η MPI παρέχει την ειδική σταθερά **MPI_ANY_TAG** την οποία μπορούμε να μεταβιβάζουμε στη θέση του ορίσματος tag της MPI_Recv. Απεναντίας οι διεργασίες – αποστολείς πρέπει να καθορίζουν συγκεκριμένο αριθμό διεργασίας – παραλήπτη και συγκεκριμένα μη αρνητική ετικέτα. Επιπρόσθετα να αναφέρουμε ότι δεν μπορεί να γίνει χρήση αυτής της ειδικής ετικέτας στα ορίσματα επικοινωνητών γιατί τόσο οι διεργασίες – αποστολείς όσο και οι διεργασίες – παραλήπτες πρέπει να καθορίζουν πάντα ρητά τον επικοινωνητή. [1]

1.6 Περισσότερα για την επικοινωνία από σημείο σε σημείο

1.6.1 Παρεμποδιστική και μη παρεμποδιστική επικοινωνία

Οι επικοινωνίες που περιγράφηκαν με τις εντολές MPI_Send και MPI_Recv είναι παρεμποδιστικές επικοινωνίες. Αυτό σημαίνει ότι δεν επιστρέφουν μέχρι να ολοκληρωθεί η επικοινωνία:

- Η **MPI_Send** δεν ολοκληρώνεται μέχρι να επαναχρησιμοποιηθεί με ασφάλεια το buffer (δηλ. το μήνυμα που θα αποσταλεί έχει αντιγραφεί έξω από το buffer)
- Η **MPI_Recv** δεν ολοκληρώνεται μέχρι να είναι έτοιμο προς χρήση το buffer (δηλαδή το μήνυμα έχει ληφθεί και αντιγραφεί στο buffer)

Η επικοινωνία δεν είναι σημαντικός χρήστης των CPU κύκλων, αλλά είναι συνήθως σχετικά αργή λόγω του δικτύου επικοινωνίας και της εξάρτησης από τη διαδικασία στο άλλο άκρο της επικοινωνίας. Με την παρεμποδιστική επικοινωνία, η διαδικασία περιμένει χαλαρά ενώ λαμβάνει χώρα η επικοινωνία.

Το MPI παρέχει παρεμποδιστική επικοινωνία για να επιτρέψει βελτιστοποίηση με αλληλεπικαλυπτόμενη επικοινωνία και υπολογισμό. Μια μη παρεμποδιστική επικοινωνία δημιουργείται από μία εντολή και στη συνέχεια μια μεταγενέστερη εντολή δοκιμάζει να διαπιστώσει εάν η επικοινωνία ολοκληρώθηκε. Αυτό επιτρέπει σε μια DMA (Direct memory access) μηχανή να αντιγράφει δεδομένα ενώ το πρόγραμμα κάνει άλλες εργασίες. Έτσι, η επικοινωνία χωρίζεται σε δύο λειτουργίες: τη δοκιμή έναρξης και ολοκλήρωσης. Η μη παρεμποδιστική επικοινωνία είναι ανάλογη με μια μορφή εξουσιοδότησης - ο χρήστης υποβάλλει ένα αίτημα στο MPI για επικοινωνία και ελέγχει ότι το αίτημά του ολοκληρώθηκε ικανοποιητικά μόνο όταν χρειάζεται να ξέρει για να προχωρήσει.

Η ρουτίνα MPI_Isend ξεκινά τη λειτουργία της μη παρεμποδιστικής αποστολής:
int MPI_Isend (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

Τα ορίσματα είναι τα ίδια όπως και για το MPI_Send με την προσθήκη ενός επιπλέον, που ονομάζεται αίτημα (request). Αυτό το όρισμα, το request, είναι πολύ σημαντικό καθώς διαθέτει ένα handle που χρησιμοποιείται για να ελέγξει πότε ολοκληρώθηκε η επικοινωνία (δηλαδή πότε μπορεί να επαναχρησιμοποιηθεί το buffer). Οι ρουτίνες MPI_Send και MPI_Isend συμπεριφέρονται με παρόμοιο τρόπο, μόνο που η τελευταία επιστρέφει αμέσως, προτού ακόμη ολοκληρωθεί η διαδικασία στη λειτουργία επικοινωνίας. Δηλαδή, το buffer

που περιέχει το μήνυμα που πρέπει να αποσταλεί δεν πρέπει να τροποποιηθεί μέχρι την ολοκλήρωση της διαδικασίας, καλώντας μια άλλη κατάλληλη ρουτίνα MPI.

Ομοίως, το MPI_Irecv ξεκινά τη μη-παρεμποδιστική λειτουργία λήψης ως εξής:

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

Έχει ένα επιπλέον όρισμα, το handle (request) για να ελέγξει την ολοκλήρωση, ακριβώς όπως κάνει το MPI_Isend. Ωστόσο, έχει επίσης ένα λιγότερο όρισμα: το status όρισμα, το οποίο χρησιμοποιείται για την επιστροφή πληροφοριών σχετικά με την ολοκληρωμένη λήψη, το οποίο όμως διαγράφεται από τη λίστα των ορισμάτων. Η λειτουργία επιστρέφει αμέσως, προτού φτάσει το μήνυμα και τοποθετηθεί στο buffer. Ο προγραμματιστής πρέπει να δοκιμάσει με μια κατάλληλη ρουτίνα MPI αν το buffer είναι έτοιμο για χρήση ή όχι.

Το MPI παρέχει δύο βασικές ρουτίνες για να ελέγξει εάν έχει ολοκληρωθεί η μη-παρεμποδιστική λειτουργία αποστολής ή λήψης. Η πρώτη είναι η MPI_Wait:

int MPI_Wait(MPI_Request *request, MPI_Status *status)

Αυτή η ρουτίνα μπλοκάρει μέχρι να ολοκληρωθεί η επικοινωνία που καθορίζεται από το αίτημα χειρισμού. Το αίτημα χειρισμού θα έχει επιστραφεί από μια προηγούμενη κλήση σε μια ρουτίνα μη-παρεμποδιστικής επικοινωνίας. Εάν η μη-παρεμποδιστική λειτουργία είναι μια ρουτίνα λήψης, τότε το status όρισμα επιστρέφει τις πληροφορίες για την ολοκληρωμένη λήψη με την ίδια μορφή που κάνει το MPI_Recv για μια παρεμποδιστική λήψη.

Η δεύτερη βασική ρουτίνα δοκιμής για μη-παρεμποδιστική επικοινωνία είναι η MPI_Test:

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

Αυτή η ρουτίνα επιστρέφει αμέσως μετά τη τιμή της μεταβλητής σε αληθή, αν έχει ολοκληρωθεί η ενέργεια που εντοπίστηκε κατόπιν αιτήματος ή σε ψευδή αν δεν έχει. Εάν η λειτουργία επικοινωνίας είναι μια λήψη και η τιμή είναι αληθής μετά την επιστροφή του MPI_Test, τότε το status περιέχει ακριβώς τις ίδιες πληροφορίες με το MPI_Recv.

Σε πολλές περιπτώσεις, κάποιος μπορεί να επιθυμεί να χρησιμοποιήσει τις εντολές test ή wait σε πολλές μη -παρεμποδιστικές λειτουργίες ταυτόχρονα. Το MPI παρέχει έναν τρόπο χρήσης της wait για όλες ή οποιεσδήποτε από τις μη-παρεμποδιστικές λειτουργίες (με MPI_Waitall και MPI_Waitany) και χρήση της test για όλες ή οποιεσδήποτε από τις μη-παρεμποδιστικές λειτουργίες (με MPI_Testall και MPI_Testany). [9]

1.6.2 Καταστάσεις επικοινωνίας

Υπάρχουν τέσσερις διαφορετικές καταστάσεις επικοινωνίας για τη λειτουργία αποστολής: Τυπική(Send), Buffered (Bsend), Σύγχρονη (Ssend) και Ready (Rsend). Μέχρι στιγμής χρησιμοποιούμε μόνο τις τυπικές και τις buffered ρουτίνες αποστολής. Και οι τέσσερις τρόποι λειτουργίας υπάρχουν και στις δύο μορφές επικοινωνίας παρεμποδιστικές και μη-παρεμποδιστικές. Στις παρεμποδιστικές μορφές, η επιστροφή από τη ρουτίνα προϋποθέτει την ολοκλήρωση. Στις μη-παρεμποδιστικές μορφές, όλες οι λειτουργίες ελέγχονται για ολοκλήρωση με τις ίδιες MPI ρουτίνες (MPI_Test, MPI_Wait κ.λπ). Κάποιος μπορεί να πάρει το όνομα της μη παρεμποδιστικής έκδοσης μίας παρεμποδιστικής ρουτίνας αποστολής απλά εισάγοντας το γράμμα «I» στο μπροστινό μέρος του αρχικού ονόματος.

SEND mode	Blocking	Non-blocking
Standard	**MPI_Send**	**MPI_Isend**
Buffered	**MPI_Bsend**	**MPI_Ibsend**
Synchronous	**MPI_Ssend**	**MPI_Issend**
Ready	**MPI_Rsend**	**MPI_Irsend**

Πίνακας 3: Ρουτίνες αποστολής της MPI [9]

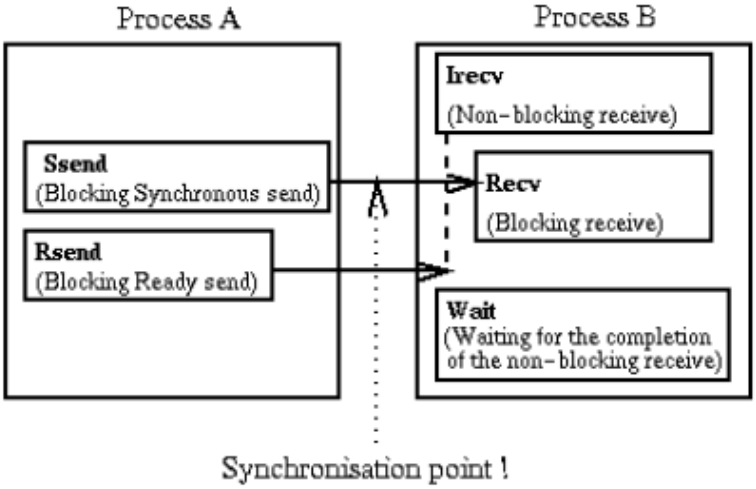
Οι τυπικές, σύγχρονες, ready και buffered αποστολές διαφέρουν μόνο ως προς τον τύπο αποστολής και όλες τους έχουν την ίδια λίστα ορισμάτων. Δίνουμε μια σύντομη περιγραφή των διαφορετικών τύπων αποστολής παρακάτω.

- **Τυπική αποστολή.** (MPI_Send, MPI_Isend) Η τυπική αποστολή ολοκληρώνεται όταν το buffer μπορεί να επαναχρησιμοποιηθεί (δηλ. το μήνυμα που αποστέλλεται να αντιγράφεται από το buffer). Η ολοκλήρωση μπορεί ή όχι να σημαίνει ότι το μήνυμα έχει φθάσει στον προορισμό του. Το μήνυμα μπορεί αντίθετα να βρίσκεται στα buffer του συστήματος για κάποιο χρονικό διάστημα. Καθώς η προσωρινή αποθήκευση του συστήματος είναι έξω από την σκοπιά του MPI, ο προγραμματιστής δεν μπορεί να υποθέσει τίποτα για την ύπαρξη ή μη τέτοιων buffer συστημάτων.
- **Buffered αποστολή**. (MPI_Bsend, MPI_Ibsend) Η Buffered αποστολή εγγυάται την άμεση ολοκλήρωση, αντιγράφοντας το μήνυμα στο buffer που παρέχεται από τον προγραμματιστή για μεταγενέστερη μετάδοση, αν είναι απαραίτητο. Το πλεονέκτημα έναντι της τυπικής αποστολής είναι η προβλεψιμότητα - ο αποστολέας και ο δέκτης εγγυώνται ότι δε θα συγχρονίζονται ακόμη και αν δεν

υπάρχει καθόλου buffer. Το μειονέκτημα της buffered αποστολής είναι ότι ο προγραμματιστής πρέπει να συνδέσει ρητά αρκετό χώρο για το πρόγραμμα με κλήσεις προς την MPI_Buffer_attach. Η μη παρεμποδιστική buffered αποστολή δεν έχει κανένα πλεονέκτημα σε σχέση με την παρεμποδιστική αποστολή.

- **Σύγχρονη αποστολή.** (MPI_Ssend, MPI_Issend) Η σύγχρονη αποστολή δεν ολοκληρώνεται μέχρι να αρχίσει η αντίστοιχη λήψη. Εάν μια διαδικασία που εκτελεί μια παρεμποδιστική σύγχρονη αποστολή είναι "μπροστά" της διεργασίας που εκτελεί την αντίστοιχη λήψη, τότε θα είναι αδρανής έως ότου η διαδικασία λήψης τη φτάσει. Παρομοίως, αν η διαδικασία αποστολής εκτελεί μια μη παρεμποδιστική σύγχρονη αποστολή, η δοκιμή ολοκλήρωσης δεν θα πετύχει μέχρι να φτάσει η διαδικασία λήψης. Επομένως, η σύγχρονη λειτουργία μπορεί να είναι πιο αργή από την τυπική λειτουργία, ωστόσο, είναι ασφαλέστερη μέθοδος επικοινωνίας, επειδή το δίκτυο επικοινωνίας δεν μπορεί ποτέ να υπερφορτωθεί με ανεπιθύμητα μηνύματα. Έχει το πλεονέκτημα έναντι του τυπικού τρόπου να είναι πιο προβλέψιμο: μια σύγχρονη αποστολή συγχρονίζει πάντα τον αποστολέα και τον παραλήπτη, ενώ μια τυπική αποστολή μπορεί ή όχι να το κάνει.
- **Ready αποστολή.** (MPI_Rsend, MPI_Irsend) Μια ready αποστολή, όπως η buffered αποστολή, ολοκληρώνεται αμέσως. Η επικοινωνία είναι εγγυημένο ότι θα επιτύχει κανονικά εάν έχει ήδη αναρτηθεί μια αντίστοιχη λήψη. Ωστόσο, σε αντίθεση με όλες τις άλλες αποστολές, αν δεν έχει αναρτηθεί η αντίστοιχη λήψη, το αποτέλεσμα δεν είναι προσδιορισμένο (π.χ. μπορεί να παρουσιαστεί σφάλμα). Έτσι, ο προγραμματιστής πρέπει να εγγυηθεί ότι έχει καταγραφεί η αντίστοιχη λήψη. Η ιδέα είναι ότι αποφεύγοντας την ανάγκη να ελέγξουμε αν η αντίστοιχη λήψη έχει ήδη καταχωρηθεί και αποφεύγοντας την αποθήκευση μεταξύ του αποστολέα και του παραλήπτη, η απόδοση μπορεί να βελτιωθεί. Η χρήση της ready λειτουργίας είναι ασφαλής μόνο αν το επιτρέπει η λογική ροή ελέγχου του παράλληλου προγράμματος. Για παράδειγμα, στο παρακάτω σχήμα (Σχήμα9), το σημείο συγχρονισμού - η παρεμποδιστική σύγχρονη αποστολή στη Διαδικασία Α - διασφαλίζει ότι η αντίστοιχη λήψη έχει ήδη καταχωρηθεί όταν η διεργασία Α καλεί τη ready λειτουργία αποστολής. Η μη παρεμποδιστική ready αποστολή δεν έχει κανένα πλεονέκτημα σε σχέση με την παρεμποδιστική ready αποστολή.

Ενώ οι λειτουργίες αποστολής έχουν τέσσερις διαφορετικές λειτουργίες επικοινωνίας, υπάρχει μόνο μία κατάσταση λήψης. Δηλαδή, τα μηνύματα που αποστέλλονται από διαφορετικά είδη αποστολών πρέπει να λαμβάνονται από τις τυπικές ρουτίνες λήψης: MPI_Recv ή MPI_Irecv(δες παρακάτω πίνακα). Οι λειτουργίες λήψης δεν ολοκληρώνονται μέχρι να φτάσει το μήνυμα και να τοποθετηθεί στο buffer. Η παρεμποδιστική λήψη δεν επιστρέφει μέχρι να ολοκληρωθεί η λειτουργία επικοινωνίας, ενώ η μη παρεμποδιστική λήψη επιστρέφει αμέσως και ο προγραμματιστής πρέπει να ελέγξει την ολοκλήρωση καλώντας μία κατάλληλη MPI λειτουργία (π.χ. MPI_Test, MPI_Wait, κλπ.).



Σχήμα 9: Παράδειγμα ασφαλής χρήσης της ready κατάστασης [9]

Mode	Blocking	Non-blocking
Standard	**MPI_Recv**	**MPI_Irecv**

Πίνακας 4: MPI ρουτίνες λήψης[9]

1.7 Συλλογική επικοινωνία

Όπως αναφέρεται στο [5], το MPI παρέχει μια ποικιλία ρουτινών για κατανεμημένα δεδομένα, για συλλογή δεδομένων, εκτέλεση καθολικών αθροίσεων κλπ. Αυτή η κατηγορία ρουτινών περιλαμβάνει τις λεγόμενες ρουτίνες "συλλογικής επικοινωνίας", αν κι ένας καλύτερος όρος μπορεί να είναι "συλλογικές λειτουργίες". Αυτό που διακρίνει τη συλλογική

επικοινωνία από την επικοινωνία από σημείο σε σημείο είναι ότι περιλαμβάνει πάντα κάθε διεργασία στον καθορισμένο επικοινωνητή (με το οποίο εννοούμε κάθε διεργασία στην ομάδα που σχετίζεται με τον επικοινωνητή). Για να εκτελέσουμε μια συλλογική επικοινωνία σε ένα υποσύνολο διεργασιών ενός επικοινωνητή, πρέπει να δημιουργηθεί ένας νέος επικοινωνητής. Τα χαρακτηριστικά της συλλογικής επικοινωνίας είναι:

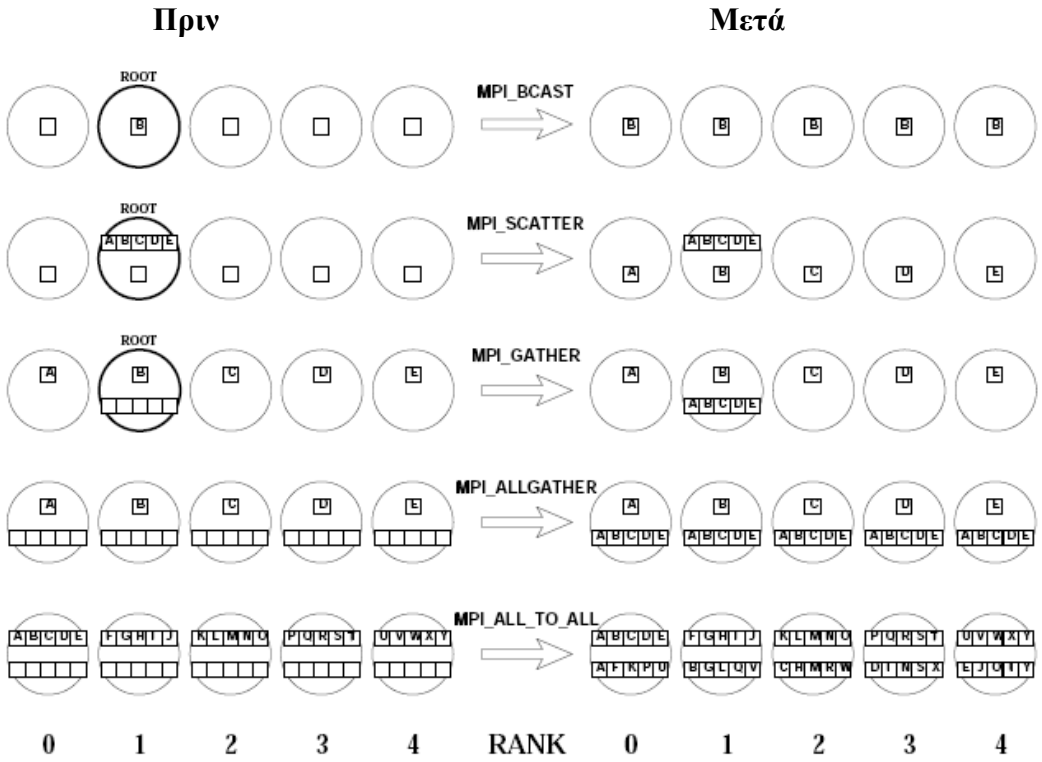
- Οι συλλογικές επικοινωνίες δεν μπορούν να επηρεάσουν τις επικοινωνίες από σημείο σε σημείο και το αντίστροφο. Η συλλογική επικοινωνία και η επικοινωνία από σημείο σε σημείο είναι διαφανείς μεταξύ τους. Για παράδειγμα, μια συλλογική επικοινωνία δεν μπορεί να ληφθεί από μια λήψη από σημείο σε σημείο. Είναι σαν να έχει κάθε επικοινωνία δύο υπο-επικοινωνίες, μία για σημείο-προς-σημείο και μία για συλλογική επικοινωνία.
- Μια συλλογική επικοινωνία μπορεί ή όχι να συγχρονίσει τις σχετικές διεργασίες.
- Ως συνήθως, η ολοκλήρωση σημαίνει ότι το buffer μπορεί να χρησιμοποιηθεί ή να ξαναχρησιμοποιηθεί. Ωστόσο, δεν υπάρχει κάτι τέτοιο, όπως η μη αποκλειστική συλλογική επικοινωνία στο MPI.
- Όλες οι διεργασίες στον επικοινωνητή πρέπει να καλέσουν τη συλλογική επικοινωνία. Ωστόσο, ορισμένα από τα ορίσματα της συνάρτησης δεν είναι σημαντικά για ορισμένες διεργασίες και μπορούν να οριστούν ως "ψεύτικες" τιμές (γεγονός που κάνει κάποιες από τις κλήσεις να φαίνονται λίγο δύσκολες!).
- Οι ομοιότητες με την επικοινωνία από σημείο σε σημείο περιλαμβάνουν:
    - Ένα μήνυμα είναι ένας πίνακας ενός συγκεκριμένου τύπου δεδομένων.
    - Οι τύποι δεδομένων πρέπει να ταιριάζουν μεταξύ της αποστολής και της λήψης.
- Οι διαφορές περιλαμβάνουν:
    - Δεν υπάρχει η έννοια των ετικετών.
    - Το απεσταλμένο μήνυμα πρέπει να γεμίσει την καθορισμένη προσωρινή μνήμη λήψης.

1.7.1 Συγχρονισμός με χρήση του Barrier (MPI_BARRIER (COMM))

Αυτή είναι η απλούστερη από όλες τις συλλογικές λειτουργίες και δεν περιλαμβάνει καθόλου δεδομένα. Το MPI_BARRIER αποκλείει την καλούσα διεργασία μέχρι να το καλέσουν όλα τα άλλα μέλη της ομάδας.

Σε μία φάση υπολογισμού, όλες οι διεργασίες συμμετέχουν στη σύνταξη ενός αρχείου. Το αρχείο πρόκειται να χρησιμοποιηθεί ως δεδομένο εισόδου για την επόμενη φάση του υπολογισμού. Επομένως, καμία διεργασία δεν πρέπει να προχωρήσει στη δεύτερη φάση μέχρις ότου όλες οι διαδικασίες να έχουν ολοκληρώσει την πρώτη φάση.

1.7.2 Broadcast, scatter, gather, etc.



Σχήμα 10: Σχηματική απεικόνιση των λειτουργιών εκπομπής / διασκορπισμού / συλλογής. Οι κύκλοι αντιπροσωπεύουν διεργασίες με ranks όπως φαίνεται. Τα μικρά πλαίσια αντιπροσωπεύουν τον χώρο προσωρινής αποθήκευσης και τα γράμματα αντιπροσωπεύουν στοιχεία δεδομένων. Οι buffer λήψης αντιπροσωπεύονται από τα κενά πλαίσια στην πλευρά "πριν" ενώ οι buffer αποστολής από τα γεμάτα πλαίσια [7].

Αυτό το σύνολο ρουτινών κατανέμει και ανακατανέμει τα δεδομένα χωρίς να εκτελεί οποιαδήποτε ενέργεια στα δεδομένα. Οι ρουτίνες παρουσιάζονται σχηματικά στο παραπάνω σχήμα: Το πλήρες σύνολο ρουτινών έχει ως εξής, ταξινομημένο εδώ σύμφωνα με τη μορφή της κληθείσης ρουτίνας.

MPI_BCAST

MPI_BCAST (buffer, count, datatype, root, comm)

Μια εκπομπή έχει μια καθορισμένη root διεργασία και κάθε διεργασία λαμβάνει ένα αντίγραφο του μηνύματος από το root. Όλες οι διεργασίες πρέπει να προσδιορίζουν το ίδιο root (και τον επικοινωνητή).

Το βασικό όρισμα είναι η τάξη της root διεργασίας. Τα ορίσματα buffer, count και datatype αντιμετωπίζονται όπως σε μία αποστολή από σημείο σε σημείο στο root και όπως και σε μία λήψη από σημείο σε σημείο οπουδήποτε αλλού.

MPI_SCATTER, MPI_GATHER

MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

MPI_GATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Αυτές οι συναρτήσεις καθορίζουν επίσης μια root διεργασία και όλες οι διεργασίες πρέπει να καθορίζουν το ίδιο root (και επικοινωνητή). Η κύρια διαφορά από το MPI_BCAST είναι ότι οι λεπτομέρειες αποστολής και λήψης διαφέρουν γενικά και πρέπει να αποσαφηνιστούν και οι δύο στις λίστες των ορισμάτων.

Να σημειώσουμε ότι το sendcount (στο root) είναι ο αριθμός των στοιχείων που στέλνονται σε κάθε διεργασία, και όχι ο αριθμός στο σύνολο. (Επομένως, αν ισχύει ότι το sendtype είναι ίσο με recvtype, τότε ισχύει ότι και το sendcount είναι ίσο με το recvcount). Το βασικό(root) όρισμα είναι η τάξη της root διεργασίας. Όπως αναμένεται, για το MPI_SCATTER, τα ορίσματα sendbuf, sendcount και sendtype είναι σημαντικά μόνο στο root (ενώ το ίδιο ισχύει και για τα ορίσματα recvbuf, recvcount και recvtype στο MPI_GATHER).

MPI_ALLGATHER, MPI_ALLTOALL

MPI_ALLGATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

MPI_ALLTOALL (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

Αυτές οι συναρτήσεις δεν έχουν καθορισμένη root διεργασία. Οι λεπτομέρειες αποστολής και λήψης είναι σημαντικές σε όλες τις διεργασίες και μπορεί να είναι διαφορετικές, οπότε και οι δύο διευκρινίζονται στις λίστες των ορισμάτων.

MPI_SCATTERV, MPI_GATHERV, MPI_ALLGATHERV, MPI_ALLTOALLV

Αυτές είναι επαυξημένες εκδόσεις των συναρτήσεων MPI_SCATTER, MPI_GATHER, MPI_ALLGATHER και MPI_ALLTOALL αντίστοιχα. Για παράδειγμα, στο MPI_SCATTERV, το όρισμα sendcount γίνεται ένας πίνακας από sendcounts, επιτρέποντας την αποστολή διαφορετικού αριθμού στοιχείων σε κάθε διεργασία. Επιπλέον,

προστίθεται ένας νέος πίνακας ακέραιων ως όρισμα (displs), το οποίο καθορίζει μετατοπίσεις, έτσι ώστε τα δεδομένα που διασκορπίζονται να μην βρίσκονται συνεχόμενα στον χώρο μνήμης της root διεργασίας. Αυτό είναι χρήσιμο για την αποστολή υπο-μπλοκ πινάκων, για παράδειγμα, και παρακάμπτει την ανάγκη να δημιουργηθεί ένας προσωρινός τύπος δεδομένων.

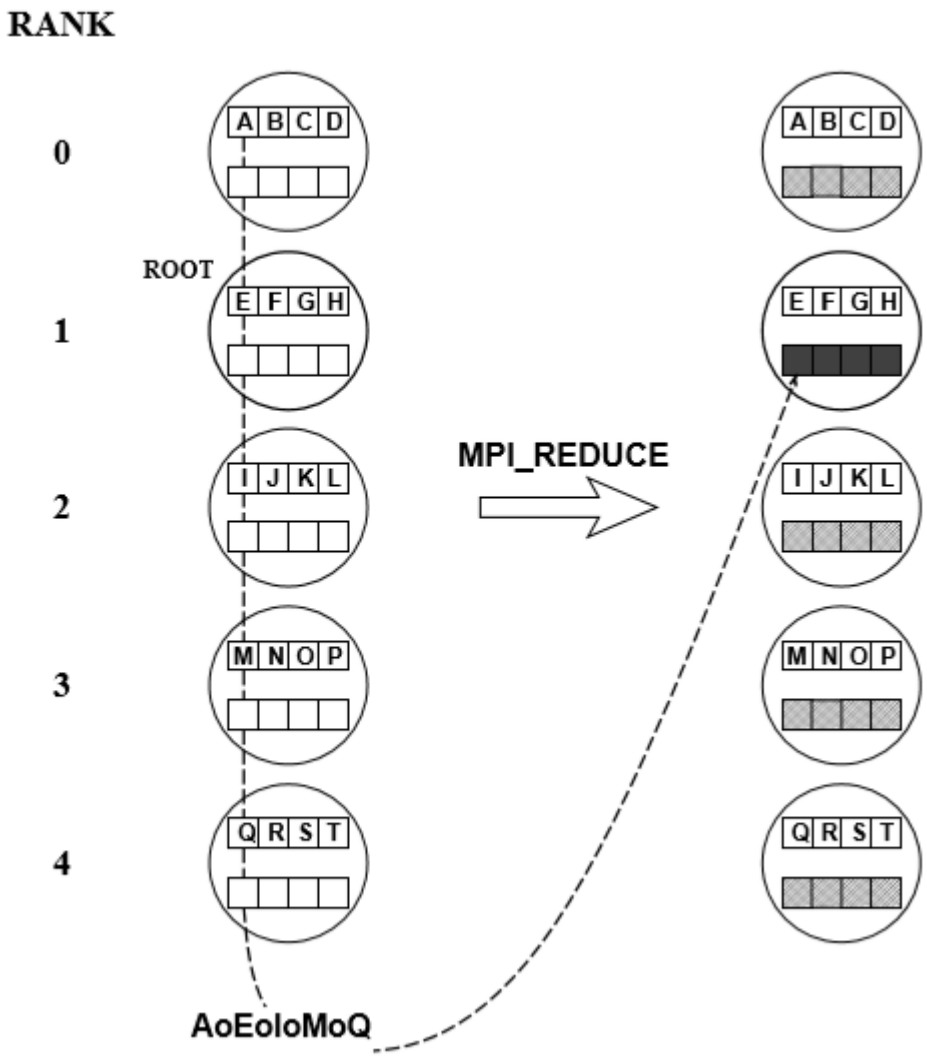
1.7.3 Καθολικές λειτουργίες μείωσης (καθολικά αθροίσματα κ.λπ.)

1.7.3.1 Πότε πρέπει να χρησιμοποιήσουμε μια καθολική λειτουργία μείωσης

Θα πρέπει να χρησιμοποιούμε συναρτήσεις καθολικής μείωσης όταν πρέπει να υπολογίσουμε ένα αποτέλεσμα το οποίο περιλαμβάνει δεδομένα που διανέμονται σε μια ολόκληρη ομάδα διεργασιών. Για παράδειγμα, αν κάθε διεργασία έχει έναν ακέραιο αριθμό, μπορεί να χρησιμοποιηθεί συνολική μείωση για να υπολογιστεί το συνολικό άθροισμα, η μέγιστη τιμή ή ο βαθμός της διεργασίας με τη μέγιστη τιμή. Ο χρήστης μπορεί επίσης να ορίσει αυθαίρετα τους πολύπλοκους τελεστές του.

1.7.3.2 MPI_REDUCE

MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm)



Σχήμα 11: Καθολική μείωση στο MPI με MPI_REDUCE. Το ο αντιπροσωπεύει τον τελεστή μείωσης. Οι κύκλοι αντιπροσωπεύουν διεργασίες με τάξεις όπως φαίνεται. Τα μικρά πλαίσια αντιπροσωπεύουν τον χώρο προσωρινής αποθήκευσης και τα γράμματα αντιπροσωπεύουν τα στοιχεία δεδομένων. Μετά την κλήση συνάρτησης, τα κουτάκια που φαίνονται ελαφρώς σκιασμένα αντιπροσωπεύουν τον χώρο προσωρινής αποθήκευσης με αδιευκρίνιστα περιεχόμενα ενώ τα σκούρα κουτάκια αντιπροσωπεύουν το αποτέλεσμα στο root. Απεικονίζεται μόνο ένα από τα τέσσερα αποτελέσματα, δηλαδή A o E o I o M o Q, αλλά τα άλλα τέσσερα είναι παρόμοια --- για παράδειγμα, το επόμενο στοιχείο του αποτελέσματος είναι B o F o J o N o R. Οι buffers λήψης αντιπροσωπεύονται από τα κενά πλαίσια στην πλευρά "πριν " ενώ οι buffers λήψης από τα γεμάτα πλαίσια.[7]

Όλες οι διεργασίες στον επικοινωνητή πρέπει να καλούν με όμοια ορίσματα εκτός από τα sendbuf και recvbuf. Να σημειωθεί ότι η root διεργασία τελειώνει με έναν πίνακα αποτελεσμάτων - αν, για παράδειγμα, αναζητηθεί ένα συνολικό άθροισμα, το root πρέπει να εκτελέσει το τελικό άθροισμα.

Κεφάλαιο 2

MPI με Python

2.1 Τι είναι η Python;

Πρόκειται για μια γλώσσα προγραμματισμού δυναμική και σύγχρονη, ιδιαίτερα εύκολη στην εκμάθηση, με μια κοινότητα χρηστών που συνεχώς αυξάνεται. Κατέχει εντυπωσιακές δομές δεδομένων υψηλού επιπέδου καθώς και μία δραστική προσέγγιση στον αντικειμενοστραφή προγραμματισμό με δυναμικό έλεγχο τύπων και δυναμική σύνδεση που είναι σχετικά απλή. Η καλαίσθητη σύνταξη της Python, μαζί με την διερμηνευόμενη φύση της, την καθιστούν ιδανική γλώσσα για δημιουργία σεναρίου εντολών(scripting) και γρήγορη ανάπτυξη εφαρμογών σε πολλούς τομείς και στις περισσότερες πλατφόρμες.[17]

Ο διερμηνέας της Python και η τεράστια βιβλιοθήκη της είναι διαθέσιμα σε πηγαία (source) ή δυαδική (binary) μορφή χωρίς κάποια χρέωση για όλες τις μεγάλες πλατφόρμες και μπορούν ελεύθερα να διανεμηθούν. Μπορεί κανείς να προσθέσει σε αυτή νέα χαρακτηριστικά, λειτουργίες και τύπους δεδομένων που υλοποιούνται σε C ή C ++. Είναι επίσης κατάλληλη ως γλώσσα επέκτασης για εφαρμογές που απαιτούν προγραμματιζόμενη διεπαφή.

Η Python είναι μία ιδανική επιλογή για τη συγγραφή των υψηλού επιπέδου μερών μεγάλων επιστημονικών εφαρμογών και προσομοιώσεων οδήγησης σε παράλληλες αρχιτεκτονικές. Οι Python κώδικες αναπτύσσονται γρήγορα, διατηρούνται εύκολα και

μπορούν να επιτύχουν υψηλό βαθμό ολοκλήρωσης με άλλες βιβλιοθήκες γραμμένες σε μεταγλωττισμένες γλώσσες.[13]

2.2 Γιατί MPI για Python (mpi4py);

Αν και οι πιο κατάλληλες γλώσσες για τον παράλληλο υπολογισμό υπολογιστών είναι η C ή η Fortran, πολλοί ερευνητές προτιμούν τις γλώσσες που βασίζονται σε πίνακες (όπως η Python ή η Matlab) αντί των C και Fortran για την ευκολία χρήσης τους. Συγκεκριμένα, η Python έχει γίνει όλο και πιο δημοφιλής. Ωστόσο, για τον ίδιο λόγο που η Python είναι μια εξαιρετική γλώσσα για τον αλγοριθμικό σχεδιασμό και για την επίλυση προβλημάτων που δεν απαιτούν μέγιστη απόδοση, αποτελεί ένα εξαιρετικό εργαλείο για την ανάπτυξη παράλληλου κώδικα. Σε γενικές γραμμές, ο παράλληλος προγραμματισμός είναι πολύ πιο δύσκολος και πολύπλοκος απ' ό,τι ο σειριακός. Αυτό κάνει τον παράλληλο προγραμματισμό στην Python τέλειο για πρωτοτυποποίηση και κατάλληλο για μικρούς έως και μεσαίους κώδικες. Μπορεί ακόμη και να θεωρηθεί κατάλληλος για παραγωγή εάν η επικοινωνία δεν είναι πολύ συχνή και οι επιδόσεις δεν είναι το κύριο μέλημα.[12]

Για να εκτελέσει κανείς παράλληλα προγράμματα στο περιβάλλον της python, πρέπει να χρησιμοποιήσει ένα module που ονομάζεται MPI4py που σημαίνει "MPI για Python". Αυτό το άρθρωμα παρέχει κοινές λειτουργίες για εκτέλεση εργασιών, όπως για παράδειγμα η εύρεση της τιμής του rank των επεξεργαστών και η αποστολή και παραλαβή μηνυμάτων μεταξύ τους.[15]

Στην παρούσα εργασία θα χρησιμοποιήσουμε την Python σε συνδυασμό με το mpi4py για παιδαγωγικούς σκοπούς. Το mpi4py φαίνεται να διατηρεί την περισσότερη λειτουργικότητα των εφαρμογών C του MPI από τα πολλά παράλληλα πακέτα της Python.

Να σημειωθεί ότι μία από τις κύριες διαφορές μεταξύ του mpi4py και του Mpi στη C ή τη Fortan, είναι ότι το mpi4py είναι αντικειμενοστραφές, εκτός από το ότι βασίζεται σε πίνακες. Ο MPI επικοινωνητής στο mpi4py είναι μια Python κλάση και οι λειτουργίες MPI όπως οι Send ή Broadcast είναι στιγμιότυπα μεθόδων της κλάσης Communicator. Θα δούμε λειτουργίες όπως η Send(…) να παρουσιάζεται ως Comm.Send(…) που σημαίνει ότι το Comm είναι ένα στιγμιότυπο της κλάσης Comm.

Επίσης, να σημειωθεί ότι το mpi4py για κάθε βασική εντολή MPI διαθέτει δύο μεθόδους επικοινωνίας. Αυτές είναι οι γνωστές εντολές με μικρά και κεφαλαία γράμματα (π.χ. Send(...) και send(...)). Οι υλοποιήσεις με κεφαλαία γράμματα χρησιμοποιούν παραδοσιακούς τύπους δεδομένων MPI κάνοντας τη χρήση του mpi4py πιο γρήγορη, ενώ οι υλοποιήσεις με μικρά γράμματα χρησιμοποιούν την pickling μέθοδο της Python που προσφέρει μεγαλύτερη ευκολία.[12]

2.3 Εισαγωγή στο MPI

2.3.1 Εγκατάσταση πακέτου

Αν έχουμε ήδη ένα MPI που λειτουργεί (είτε το εγκαταστήσαμε από πηγές είτε χρησιμοποιώντας ένα προ-κατασκευασμένο πακέτο από την αγαπημένη μας διανομή GNU / Linux) και ο mpicc μεταγλωττιστής βρίσκεται στη διαδρομή αναζήτησης, τότε μπορούμε να χρησιμοποιήσουμε την εντολή pip:

$ [sudo] pip install mpi4py

2.3.2 Hello World

Όπως συνήθως, θα κάνουμε μια εισαγωγή στον MPI python προγραμματισμό χρησιμοποιώντας μια παραλλαγή στο πρότυπο πρόγραμμα hello world: το πρώτο μας MPI πρόγραμμα θα είναι το πρόγραμμα Hello World για πολλαπλές διεργασίες. Ο πηγαίος κώδικας έχει ως εξής:

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
3 rank = comm.Get_rank()
4 size = comm.Get_size()
5 print('Hello world from process {} out of {}'.format(rank, size))
```

Αφού αποθηκεύσουμε αυτόν τον κώδικα ως helloworld.py, τον εκτελούμε χρησιμοποιώντας την ακόλουθη σύνταξη γραμμής εντολών, που τρέχει από τον κατάλογο του αρχείου:

```
$ mpiexec  -n 5 python  helloworld.py
```

Η παραπάνω εντολή θα εκτελέσει το πρόγραμμα πέντε φορές γιατί πέντε είναι οι διεργασίες που θα το εκτελέσουν. Όταν κάθε πρόγραμμα τρέξει, θα εκτυπώσει hello world και θα μας πει το rank της αντίστοιχης διεργασίας:

```
Hello world from process 0 out of 5
Hello world from process 1 out of 5
Hello world from process 2 out of 5
Hello world from process 3 out of 5
Hello world from process 4 out of 5
```

Αν το τρέξουμε κάμποσες φορές θα διαπιστώσουμε ότι δεν εκτυπώνονται κατ 'ανάγκην στη σειρά τα αποτελέσματα. Αυτό οφείλεται στο γεγονός ότι οι 5 ξεχωριστές διεργασίες εκτελούνται σε διαφορετικούς επεξεργαστές και δεν μπορούμε να γνωρίζουμε εκ των προτέρων ποια θα εκτελέσει πρώτα την δήλωση εκτύπωσης. Εάν οι διεργασίες προγραμματίζονται στον ίδιο επεξεργαστή αντί για πολλαπλούς επεξεργαστές, τότε εξαρτάται από το λειτουργικό σύστημα να προγραμματίσει τις διεργασίες. Στην ουσία, κάθε διεργασία εκτελείται αυτόνομα.

Συνήθως είναι κακή πρακτική να εκτελούμε I / O (π.χ. κλήση εκτύπωσης) από οποιαδήποτε διεργασία εκτός της διεργασίας root, αν και μπορεί συχνά να είναι ένα χρήσιμο εργαλείο για τον εντοπισμό σφαλμάτων. Το κάνουμε εδώ, ωστόσο, για παιδαγωγικούς σκοπούς.

2.3.3 Εκτέλεση

Όπως αναφερθήκαμε και προηγουμένως, τα προγράμματα mpi4py είναι προγράμματα SPMD (single - program multiple - data) και επομένως κάθε διεργασία θα εκτελέσει τον ίδιο κώδικα λίγο διαφορετικά. Όταν εκτελούμε την παρακάτω εντολή, συμβαίνουν πολλά πράγματα:

```
$ mpiexec  -n 5 python  hello.py
```

Πρώτον, ξεκινάει το πρόγραμμα mpiexec. Αυτό είναι το πρόγραμμα που ξεκινά την MPI, ένα περιτύλιγμα γύρω από οποιοδήποτε πρόγραμμα. Η επιλογή -n 5 καθορίζει τον επιθυμητό αριθμό διεργασιών. Στην περίπτωσή μας, εκτελούνται 5 διεργασίες, κάθε μία από τις οποίες είναι ένα στιγμιότυπο της python. Σε κάθε ένα από τα 5 στιγμιότυπα της python, περνάμε το όρισμα hello.py το οποίο είναι το όνομα του αρχείου κειμένου του προγράμματος μας, που βρίσκεται στον τρέχοντα κατάλογο. Κάθε ένα από τα πέντε στιγμιότυπα της python ανοίγει στη συνέχεια το αρχείο .py και τρέχει το ίδιο πρόγραμμα. Η διαφορά στο περιβάλλον εκτέλεσης κάθε διεργασίας είναι ότι στις διεργασίες δίδονται διαφορετικές τιμές του rank στον επικοινωνητή. Εξαιτίας αυτού, κάθε διεργασία εκτυπώνει διαφορετικό αριθμό όταν εκτελείται.

Η MPI και η python συνδυάζονται έτσι ώστε να γίνει ο πηγαίος κώδικας όσο πιο σαφής και σύντομος γίνεται. Η πρώτη γραμμή του παραπάνω προγράμματος είναι ένα σχόλιο ενώ η δεύτερη γραμμή μας παρέχει από το πακέτο mpi4py το MPI module. Χρησιμοποιώντας τον τελεστή **.**(τελεία) μπορούμε να αποκτήσουμε πρόσβαση σε ένα στατικό αντικείμενο του επικοινωνητή, από όπου η τρέχουσα διεργασία μπορεί να πληροφορηθεί για την τιμή του rank της.

Το MPI.COMM_WORLD είναι μια στατική αναφορά σε ένα αντικείμενο Comm και χρησιμοποιώντας το comm ουσιαστικά αυτό που κάνουμε είναι μια απλή αναφορά σε αυτό. Θα μπορούσαμε να παραλείψουμε την τρίτη γραμμή και απλώς να χρησιμοποιούσαμε το MPI.COMM_WORLD στη θέση του comm και το πρόγραμμα δε θα άλλαζε. Ένας Communicator αντιπροσωπεύει ένα σύστημα υπολογιστών ή επεξεργαστών που μπορούν να επικοινωνούν μεταξύ τους μέσω εντολών MPI. Τα αντικείμενα Comm έχουν πολλές μεθόδους και ιδιότητες που αρκετές από αυτές δε συσχετίζονται με την προδιαγραφή MPI.[12]

2.3.4 Ο Επικοινωνητής (The Communicator)

Ένας επικοινωνητής είναι μια λογική μονάδα που καθορίζει ποιες διεργασίες επιτρέπεται να στέλνουν και να λαμβάνουν μηνύματα. Με την οργάνωση διεργασιών με αυτόν τον τρόπο, το MPI μπορεί φυσικά να αναδιατάξει ποιες διεργασίες έχουν ανατεθεί σε ποιους επεξεργαστές και να βελτιστοποιήσει το πρόγραμμα ως προς την ταχύτητα.

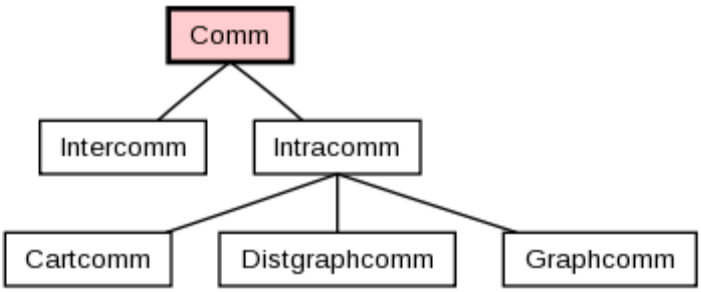
2.3.4.1 Intracommunicators and Intercommunicators

Το MPI παρέχει communicators οι οποίοι χρησιμοποιούνται για επικοινωνία εσωτερικά σε μία ομάδα (intra-communicators) και άλλους που χρησιμοποιούνται για την επικοινωνία μεταξύ διαφορετικών ομάδων (inter-communicators).[11] Στην MPI για την Python, το MPI.Comm είναι η βασική κλάση των επικοινωνητών. Οι κλάσεις MPI.Intracomm και MPI.Intercomm αποτελούν υποκλάσεις της κλάσης MPI.Comm.

Οι intracommunicators είναι η πιο συνηθισμένη μορφή επικοινωνητή στο MPI στους οποίους λαμβάνει χώρα η περισσότερη MPI επικοινωνία . Κάθε intracommunicator περιέχει ένα σύνολο διεργασιών και αναγνωρίζεται από το "rank" του μέσα στον επικοινωνητή. Τα ranks αριθμούνται από το 0 έως το Size-1. Οποιαδήποτε διεργασία στον επικοινωνητή μπορεί να στείλει ένα μήνυμα σε άλλη διεργασία εντός του επικοινωνητή ή να λάβει ένα μήνυμα από οποιαδήποτε άλλη διεργασία στον επικοινωνητή. Υποστηρίζουν επίσης μια ποικιλία συλλογικών λειτουργιών που περιλαμβάνουν όλες τις διεργασίες στον επικοινωνητή. [12] Σύμφωνα με τη βιβλιογραφία υπάρχουν δύο αντικείμενα του intracommunicator: τα MPI.COMM_SELF και MPI.COMM_WORLD. Όποτε κριθεί απαραίτητο μάλιστα τα αντικείμενα αυτά μπορούν να δημιουργήσουν καινούριους επικοινωνητές.

Ευνοούνται πλήρως οι εικονικές τοπολογίες (κλάσεις MPI.Cartcomm, MPI.Graphcomm και MPI.Distgraphcomm, που είναι υποπεριπτώσεις της κλάσης MPI.Intracomm). Επίσης με τις μεθόδους MPI.Intracomm.Create_cart() και MPI.Intracomm.Create_graph() μπορούν να ληφθούν νέα αντικείμενα από αντικείμενα intracommunicator.[14]

Η ιεραρχία των επικοινωνητών εμφανίζεται στο παρακάτω σχήμα.



Σχήμα 12: Η ιεραρχία των επικοινωνητών[12]

2.3.5 Ανακαλύπτοντας τον υπόλοιπο κόσμο

Στο mpi4py για να μάθουμε για άλλες διεργασίες θα πρέπει να χρησιμοποιήσουμε μαζί τις μεθόδους Comm.Get_rank(), και Comm.Get_Size(). Η ύπαρξη των ranks είναι πολύ σημαντική προκειμένου να ενημερωθούμε για την ύπαρξη των άλλων διεργασιών. Συγκεκριμένα, ένα rank αποτελεί το id της διεργασίας μέσα σε έναν επικοινωνητή. Αν στο πρόγραμμά μας επομένως χρησιμοποιήσουμε την εντολή Comm.Get_rank(), ουσιαστικά τότε την εντολή αυτή την καλεί κάθε διεργασία στη μεταβλητή του επικοινωνητή comm κι έτσι το rank κάθε διεργασίας αποθηκεύεται στην μεταβλητή που υποδεικνύεται από το ίδιο το rank. Το rank αναφέρεται σε μια τοπική μεταβλητή, η οποία είναι μοναδική για κάθε διεργασία που την καλεί κι αυτό επειδή κάθε διεργασία έχει το δικό της ξεχωριστό αντίγραφο τοπικών μεταβλητών. Από την άλλη, το Comm.Get_Size επιστρέφει το πλήθος των επεξεργαστών στον Communicator.

Συνεπώς, για να μπορέσει κανείς να επικοινωνήσει με μια άλλη διεργασία, το μόνο που χρειάζεται να γνωρίζει είναι το rank της άλλης διεργασίας που παίρνει τιμές από το 0 έως το comm.size - 1 μη συμπεριλαμβάνοντας όμως το rank της ίδιας της διεργασίας. [12]

2.3.6 Πιο συγκεκριμένα: Get_size() και Get_rank()

- **Comm.Get_size()**

  Επιστρέφει τον αριθμό των διεργασιών στον επικοινωνητή που είναι ίδιος σε κάθε διεργασία.

  Παράμετροι:

  Comm (*MPI comm*) – επικοινωνητής που θέλουμε να υποβάλλουμε ερώτημα

  Rvalue: αριθμός διεργασιών στον επικοινωνητή

  Τύπος επιστροφής: ακέραιος (integer)

  Παράδειγμα

```
from mpi4py import MPI
size = MPI.COMM_WORLD.Get_size()
```

- **Comm.Get_rank()**

  Επιστρέφει το rank της κληθείσας διεργασίας στον επικοινωνητή.

  Παράμετροι:

  Comm (*MPI comm*) – επικοινωνητής που θέλουμε να υποβάλλουμε ερώτημα

  Rvalue: rank της κληθείσας διεργασίας στον επικοινωνητή

  Τύπος επιστροφής: ακέραιος (integer)

  Παράδειγμα

  from mpi4py import MPI

  rank = MPI.COMM_WORLD.Get_rank()

Να σημειωθεί εκ νέου ότι μία από τις κύριες διαφορές μεταξύ του mpi4py και του MPI στη C ή στη Fortan, εκτός από το ότι βασίζεται σε πίνακες, είναι ότι το mpi4py είναι αντικειμενοστραφές. Ο MPI Επικοινωνητής στο mpi4py είναι μια κλάση Python και οι λειτουργίες MPI όπως Get_size() ή Get_rank() είναι Μέθοδοι Αντικειμένων (Instance Methods) της κλάσης Communicator. Η μέθοδος για παράδειγμα Get_rank() παρουσιάζεται ως Comm.Get_rank() όπου υπονοείται ότι το Comm είναι ένα αντικείμενο της κλάσης Comm.[12]

2.3.7 Διαφορετικοί κώδικες σε ένα αρχείο

Καθώς ένα MPI πρόγραμμα εκτελείται, κάθε διεργασία αποκτά τον ίδιο κώδικα. Όμως, κάθε διεργασία έχει διαφορετική τιμή στο rank. Αυτό μας δίνει τη δυνατότητα να ενσωματώσουμε έναν ξεχωριστό κώδικα για κάθε διεργασία σε ένα μόλις αρχείο. Στον κώδικα που ακολουθεί, σε όλες τις διεργασίες δίδονται οι ίδιοι δύο αριθμοί. Ωστόσο, αν και υπάρχει μόνο ένα αρχείο, στις τρεις διεργασίες δίδονται εντελώς διαφορετικές οδηγίες για το τι πρέπει να κάνουν με αυτούς τους αριθμούς. Η διεργασία «μηδέν» τους πολλαπλασιάζει, η διεργασία «ένα» τους προσθέτει και η διεργασία «δύο» παίρνει τη μικρότερη τιμή από αυτούς:

```
1  from mpi4py import MPI
2  rank = MPI.COMM_WORLD.Get_rank()
```

```
3
4   num1 = 5.0
5   num2= 2.0
6   if (rank == 0):
7       print num1 * num2
8   if (rank == 1):
9       print num1 + num2
10  if (rank == 2):
11      print min(num1, num2)
```

2.4 Επικοινωνία από σημείο σε σημείο

Η επικοινωνία από σημείο σε σημείο είναι μια θεμελιώδης δυνατότητα των συστημάτων που μεταδίδουν μηνύματα. Αυτός ο μηχανισμός επιτρέπει τη μετάδοση δεδομένων μεταξύ ενός ζεύγους διεργασιών όπου μια πλευρά στέλνει και η άλλη λαμβάνει.

2.4.1 Παρεμποδιστικές επικοινωνίες

Το MPI παρέχει βασικές λειτουργίες αποστολής και λήψης που παρεμποδίζουν. Αυτές οι λειτουργίες παρεμποδίζουν τον καλούντα έως ότου τα buffers των δεδομένων που εμπλέκονται στην επικοινωνία να μπορούν να επαναχρησιμοποιηθούν με ασφάλεια από το πρόγραμμα εφαρμογής.

Στην MPI για την Python, οι μέθοδοι MPI.Comm.Send(), MPI.Comm.Recv() και MPI.Comm.Sendrecv() των αντικειμένων του Communicator παίζουν καταλυτικό ρόλο για την παρεμπόδιση επικοινωνιών από σημείο σε σημείο. Οι μέθοδοι MPI.Comm.send(), MPI.Comm.recv() και MPI.Comm.sendrecv() (με μικρά) μπορούν να μεταδίδουν γενικά αντικείμενα της Python.[14]

- **Comm.Send(buf, dest = 0, tag = 0)**

  Εκτελεί μια βασική αποστολή. Αυτή η αποστολή είναι μια επικοινωνία από σημείο σε σημείο. Στέλνει πληροφορίες από ακριβώς μια διεργασία σε μια ακριβώς άλλη διεργασία.

Παράμετροι

Comm (MPI comm) – επικοινωνητής που θέλουμε να υποβάλλουμε ερώτημα
buf (choice) – δεδομένα για αποστολή. Δε χρειάζεται να έχουν μόνο σταθερή τιμή (ή rank), μπορεί να είναι μια έκφραση.
dest (integer) – rank προορισμού
tag (integer) – ετικέτα μηνύματος

Δηλαδή: comm.send ((rank*2) + 8, dest = 1).
Αυτό στέλνει το μήνυμα "(rank*2) + 8" στον κόμβο με βαθμό διεργασίας 1. Επομένως, μόνο αυτός ο κόμβος μπορεί να το λάβει.

- **Comm.Recv(buf, source = 0, tag = 0, Status status = None)**

  Αυτό υποδεικνύει μια συγκεκριμένη διεργασία για την λήψη δεδομένων / μηνύματος μόνο από τη διεργασία με βαθμό που αναφέρεται στην παράμετρο "source". Όταν ένας κόμβος εκτελεί τη μέθοδο recv(), περιμένει μέχρι να λάβει κάποια δεδομένα από τον αναμενόμενο αποστολέα. Αφού λάβει κάποια δεδομένα, συνεχίζει με το υπόλοιπο πρόγραμμα.

Παράμετροι

Comm (MPI comm) – επικοινωνητής που θέλουμε να υποβάλλουμε ερώτημα
buf (choice) – αρχική διεύθυνση του buffer λήψης
source (integer) – rank πηγής. Δε χρειάζεται να έχει μόνο σταθερή τιμή (ή rank), μπορεί να είναι μια έκφραση.
tag (integer) – ετικέτα μηνύματος
status (Status) – κατάσταση αντικειμένου

Για παράδειγμα: comm.recv (source= 1)
Σημαίνει ότι λαμβάνει το μήνυμα μόνο από μια διεργασία με βαθμό διεργασίας 1.

2.4.2 Η χρήση του tag στις send() και recv() συναρτήσεις

Όταν χρησιμοποιούμε τις ετικέτες στις συναρτήσεις send() και recv(), μπορούμε να εγγυηθούμε τη σειρά λήψης μηνυμάτων κι έτσι να είμαστε βέβαιοι ότι ένα μήνυμα θα παραδοθεί πριν από ένα άλλο.

Προκειμένου να επιτευχθεί ένα κάποιο είδος συγχρονισμού, θα πρέπει ένα συγκεκριμένο send() να ταιριάζει με ένα συγκεκριμένο recv(). Αυτό μπορεί να γίνει με τη χρήση της παραμέτρου "tag" και στο send() αλλά και στο recv().

Για παράδειγμα, ένα send() μπορεί να μοιάζει με:

com.send (data, dest = 2, tag = 1) και ένα αντίστοιχο recv() με την παραπάνω δήλωση θα έμοιαζε με: comm.recv(source=1,tag=1).

Έτσι, αυτή η δομή αναγκάζει ένα ταίριασμα, που οδηγεί σε συγχρονισμό των μεταφερόμενων δεδομένων. Το πλεονέκτημα του tagging είναι ότι ένα recv() μπορεί να περιμένει μέχρι να λάβει δεδομένα από ένα αντίστοιχο send(). Όμως, αυτή η μέθοδος πρέπει να χρησιμοποιηθεί με εξαιρετική προσοχή καθώς μπορεί να οδηγήσει σε κατάσταση αδιεξόδου.[15]

Στη συνέχεια θα δούμε με παράδειγμα τη σύνταξη των μηνυμάτων Send και Recv.

<u>Παράδειγμα</u>:

```
1  from mpi4py import MPI
2  import numpy as np
3  data = np([5,6,7])
4  if MPI.COMM_WORLD.rank == 0:
5      MPI.COMM_WORLD.Send(data, dest = 1)
6  else:
7      MPI.COMM_WORLD.Recv(data, source = 0)
```

Στη 2η γραμμή του κώδικα παρατηρούμε ότι κάνουμε import ένα πακέτο με το όνομα numpy. Το Numpy ("Numeric Python" ή "Numerical Python") είναι η βασική βιβλιοθήκη για επιστημονικούς υπολογισμούς στην Python. Διαθέτει ένα αντικείμενο πολυδιάστατου πίνακα υψηλής απόδοσης και εργαλεία για εργασία με αυτούς τους πίνακες. Ως numpy πίνακας ορίζεται ένα πλέγμα τιμών, που έχουν τον ίδιο τύπο, και ευρετηριάζεται από μια πλειάδα μη αρνητικών ακεραίων. Ο αριθμός των διαστάσεων είναι το rank του πίνακα και το shape

αυτού είναι μια πλειάδα ακεραίων που δίνει το μέγεθος του πίνακα κατά μήκος κάθε διάστασης.[22]

Η βασική βιβλιοθήκη της Python διαθέτει λίστες. Μια λίστα ισοδυναμεί με έναν πίνακα, αλλά έχει τη δυνατότητα αλλαγής του μεγέθους και μπορεί να περιέχει στοιχεία διαφορετικών τύπων. Η διαφορά των λιστών με τον numpy πίνακα είναι η απόδοση. Οι numpy δομές δεδομένων έχουν καλύτερη απόδοση σε:

- Μέγεθος - οι numpy δομές δεδομένων καταλαμβάνουν λιγότερο χώρο
- Απόδοση - έχουν ανάγκη για ταχύτητα και είναι ταχύτερες από τις λίστες
- Λειτουργικότητα - οι numpy δομές δεδομένων έχουν βελτιστοποιημένες λειτουργίες όπως ενσωματωμένες λειτουργίες γραμμικής άλγεβρας.[21]

Οι εντολές λοιπόν Send και Recv αναφέρονται ως παρεμποδιστικές λειτουργίες. Δηλαδή, αν μια διεργασία καλέσει τη Recv, θα παραμείνει αδρανής μέχρι να λάβει ένα μήνυμα από την αντίστοιχη εντολή Send πριν προχωρήσει. Υπάρχουν και οι αντίστοιχες μη παρεμποδιστικές λειτουργίες Isend και Irecv (το I σημαίνει άμεσο). Στην ουσία, το Irecv θα επιστρέψει αμέσως. Εάν δεν λάβει το μήνυμα, θα υποδείξει στο σύστημα ότι πρόκειται να λάβει ένα μήνυμα, θα προχωρήσει πέρα από το Irecv για να κάνει άλλη χρήσιμη δουλειά και στη συνέχεια θα ελέγξει εάν το μήνυμα έχει φτάσει. Αυτό μπορεί να χρησιμοποιηθεί για αισθητή βελτίωση της απόδοσης. Η παρεμποδιστική συμπεριφορά δεν είναι πάντα επιθυμητή στον παράλληλο προγραμματισμό. Σε ορισμένες περιπτώσεις είναι πιο ωφέλιμο να χρησιμοποιηθούν μέθοδοι μη-παρεμποδιστικής επικοινωνίας.[16]

Ας δούμε παρακάτω ένα παράδειγμα χρησιμοποιώντας τις ρουτίνες Comm.Send και Comm.Recv. Πρόκειται για ένα πρόγραμμα που είναι σχεδιασμένο για δύο διεργασίες. Στη μία θα επιλεχθεί τυχαία ένας αριθμός και στη συνέχεια θα σταλεί στην άλλη.

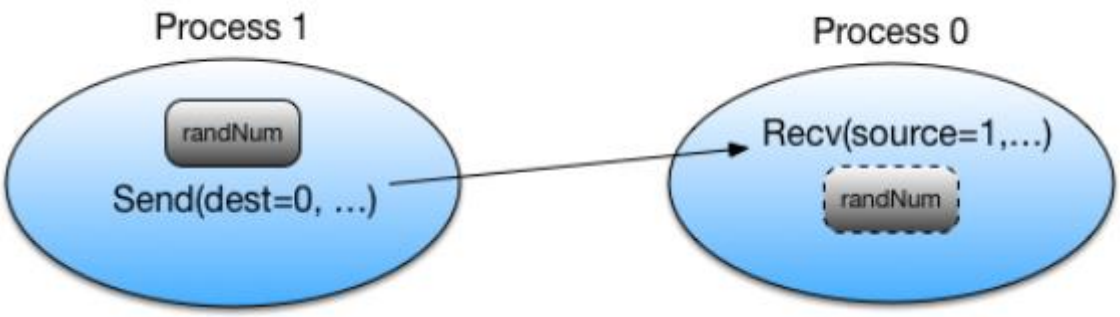
```
1  from mpi4py import MPI
2  import numpy as np
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5
6  number = np.zeros(1)
7
```

```
8   if rank == 1:
9       number = np.random.random(1)
10      print("Process", rank, "picked number", number[0])
11      comm.Send(number, dest=0)
12
13  if rank == 0:
14      print("Process", rank, "initially has the number", number[0])
15      comm.Recv(number, source=1)
16      print("Process", rank, "received the number", number[0])
```

Για να αποδείξουμε ότι ο κώδικας λειτουργεί με τον τρόπο που περιμένουμε, έχουμε τη διεργασία που εκτυπώνει την τιμή που τράβηξε τυχαία. Έπειτα, έχουμε τη διεργασία λήψης που εκτυπώνει την τιμή της μεταβλητής στην οποία θα λάβουμε για να δείξουμε ότι είναι μηδέν και στη συνέχεια εκτυπώνουμε την τιμή που λαμβάνει μετά την κλήση της Recv.

Το πρόγραμμα αυτό παράγει το ακόλουθο αποτέλεσμα:
('Process', 0, 'initially has the number', 0.0)
('Process', 1, 'picked number', 0.5320160093090147)
('Process', 0, 'received the number', 0.5320160093090147)

Εδώ βλέπουμε ότι οι λειτουργίες Send και Recv λειτουργούν παρεμποδιστικά όπως και αναφέραμε προηγουμένως. Αν μια διεργασία καλεί το Recv, απλώς θα περιμένει μέχρι να λάβει ένα μήνυμα από το αντίστοιχο Send πριν προχωρήσει. Ομοίως, το Send θα περιμένει έως ότου παραληφθεί το μήνυμα από το αντίστοιχο Recv.[19]



Σχήμα 13: Παρεμποδιστική λειτουργία[19]

Συμβουλή: Σε μία εντολή Recv δεν είναι απαραίτητο να καθορίζουμε πάντα την πηγή(source). Αντ 'αυτού, μπορούμε να επιτρέψουμε στην καλούσα διεργασία να δεχτεί ένα μήνυμα από οποιαδήποτε διεργασία που στέλνει στη διεργασία λήψης. Αυτό γίνεται με τη ρύθμιση του source σε μια προκαθορισμένη MPI σταθερά , source = ANY_SOURCE.[12,19]

Ας το δούμε σε ένα σχετικό παράδειγμα:

```
1  from mpi4py import MPI
2  import numpy as np
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5
6  number = np.zeros(1)
7
8  if rank == 1:
9          number = np.random.random(1)
10          print("Process", rank, "picked number", number[0])
11         comm.Send(number, dest=0)
12         comm.Recv(number, source=MPI.ANY_SOURCE)
13         print("Process", rank, "received the number", number[0])
14
15 if rank == 0:
14          print("Process", rank, "initially has the number", number[0])
15         comm.Recv(number, source=MPI.ANY_SOURCE)
16          print("Process", rank, "received the number", number[0])
17          number *= 2
18          comm.Send(number, dest=1)
```

Η έξοδος του προγράμματος θα μοιάζει κάπως έτσι:

('Process', 1, 'picked number', 0.4020385106367499)

('Process', 0, 'initially has the number', 0.0)

('Process', 0, 'received the number', 0.4020385106367499)

('Process', 1, 'received the number', 0.8040770212734998)

2.4.3 Μη - Παρεμποδιστικές επικοινωνίες

Σε πολλά συστήματα, η επίδοση μπορεί να αυξηθεί σημαντικά με αλληλεπικαλυπτόμενη επικοινωνία και υπολογισμό. Αυτό ισχύει ιδιαίτερα για συστήματα όπου η επικοινωνία μπορεί να εκτελείται αυτόνομα από έναν έξυπνο, εξειδικευμένο ελεγκτή επικοινωνίας.

Το MPI παρέχει μη-παρεμποδιστικές λειτουργίες αποστολής (MPI.Comm.Isend()) και λήψης (MPI.Comm.Irecv()) που επιτρέπουν την πιθανή επικάλυψη της επικοινωνίας και του υπολογισμού. Η μη-παρεμποδιστική επικοινωνία χωρίζεται σε δύο μέρη: τις posting λειτουργίες, οι οποίες ξεκινούν την αιτούμενη λειτουργία και τις test-for-completion λειτουργίες, οι οποίες μας επιτρέπουν να ανακαλύψουμε εάν η αιτούμενη λειτουργία έχει ολοκληρωθεί.[14]

Αυτές οι μέθοδοι επιστρέφουν ένα στιγμιότυπο MPI.Request, καθορίζοντας μοναδικά την έναρξη της λειτουργίας. Η ολοκλήρωσή αυτής μπορεί να γίνει με τη χρήση των μεθόδων MPI.Request.Test(), MPI.Request.Wait() και MPI.Request.Cancel(). Αν είναι απαραίτητο, μπορεί κανείς να κάνει κάποιους υπολογισμούς μεταξύ MPI.Comm.Isend() και MPI.Request.Wait() για να αυξήσει την αποδοτικότητα με την επικάλυψη των επικοινωνιών και των υπολογισμών.[16]

<u>Παράδειγμα</u>

```
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
3  rank = comm.Get_rank()
4  size = comm.Get_size()
5
6  if rank == 0:
7        data = {'a': 5, 'b': 6}
8       for i in range(1, size):
9            req = comm.isend(data, dest=i, tag=i)
10            req.wait()
11            print('Process {} sent data:'.format(rank), data)
12 else:
13          req = comm.irecv(source=0, tag=rank)
14          data = req.wait()
```

```
15          print('Process {} received data:'.format(rank), data)
```

Η έξοδος του προγράμματος είναι η ακόλουθη:

('Process 0 sent data:', {'a': 5, 'b': 6})

('Process 1 received data:', {'a': 5, 'b': 6})

2.4.4 Συνεχείς επικοινωνίες

Συχνά μια επικοινωνία που υπάρχει η ίδια λίστα ορισμάτων εκτελείται συνεχώς σε έναν εσωτερικό βρόχο. Προκειμένου να βελτιστοποιήσουμε την επικοινωνία θα πρέπει να χρησιμοποιήσουμε τη συνεχή επικοινωνία. Πρόκειται για μια ειδική περίπτωση μη παρεμποδιστικής επικοινωνίας που μειώνει το overhead μεταξύ των διεργασιών και των ελεγκτών επικοινωνίας. Επιπλέον, αυτό το είδος βελτιστοποίησης μπορεί επίσης να ανακουφίσει το πρόσθετο overhead που σχετίζεται με διερμηνευόμενες δυναμικές γλώσσες όπως η Python.

Πιο συγκεκριμένα, εδώ βρίσκουμε τις μεθόδους MPI.Comm.Send_init() και MPI.Comm.Recv_init() που συνδέονται με τη δημιουργία συνεχών αιτημάτων για τις λειτουργίες αποστολής και λήψης. Αυτές οι μέθοδοι επιστρέφουν ένα στιγμιότυπο της κλάσης MPI.Prequest, που αποτελεί υποκλάση της κλάσης MPI.Request. Η ουσιαστική επικοινωνία μπορεί να ξεκινήσει χρησιμοποιώντας τη μέθοδο MPI.Prequest.Start() και η ολοκλήρωσή της μπορεί να αντιμετωπιστεί όπως περιγράφηκε προηγουμένως.[14]

2.5 Συλλογική επικοινωνία

Οι συλλογικές επικοινωνίες επιτρέπουν ταυτόχρονα τη μετάδοση δεδομένων μεταξύ πολλών διεργασιών μιας ομάδας. Η σύνταξη και η σημασιολογία των συλλογικών λειτουργιών είναι σύμφωνη με την επικοινωνία από σημείο σε σημείο κι έρχεται μόνο σε παρεμποδιστικές εκδόσεις. Επίσης μεταδίδουν τα δεδομένα χωρίς τα μηνύματα να

συνδυάζονται με μια σχετική ετικέτα. Μάλιστα η ικανότητα διάκρισης των μηνυμάτων υπονοείται στη σειρά κλήσης.

Οι πιο συνηθισμένες διεργασίες συλλογικής επικοινωνίας που χρησιμοποιούνται είναι οι ακόλουθες.

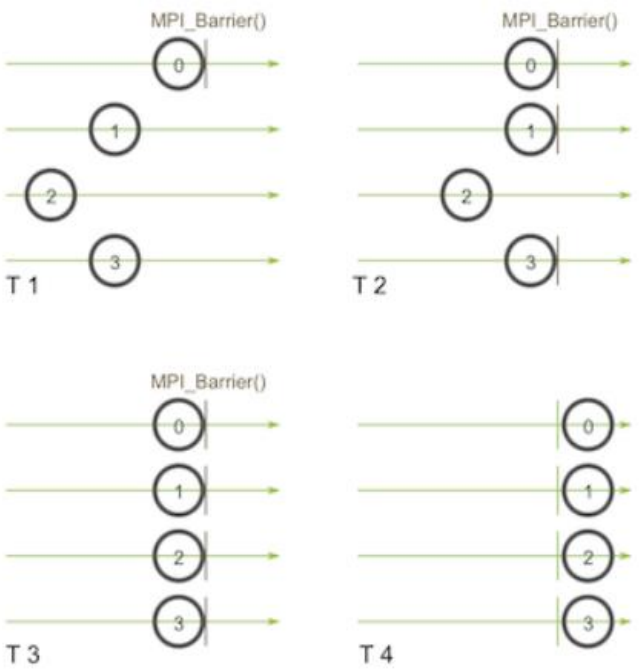
- Συγχρονισμός φραγής (barrier) σε όλα τα μέλη ομάδας.
- Καθολικές λειτουργίες επικοινωνίας
    - Εκπομπή(broadcast) μηνυμάτων από ένα μέλος σε όλα τα μέλη μιας ομάδας.
    - Συγκέντρωση(gather) δεδομένων από όλα τα μέλη σε ένα μέλος μιας ομάδας.
    - Διασπορά(scatter) δεδομένων από ένα μέλος σε όλα τα μέλη μιας ομάδας.
- Καθολικές λειτουργίες μείωσης(reduction) όπως το άθροισμα, το μέγιστο, το ελάχιστο, κλπ.[14]

2.5.1 Συγχρονισμός φραγής (barrier) σε όλα τα μέλη ομάδας.

Στη συλλογική επικοινωνία υπονοείται η ύπαρξη ενός σημείου συγχρονισμού μεταξύ των διεργασιών που όταν φτάσουν όλες σε αυτό το σημείο στον κώδικα τους να μπορέσουν να ξεκινήσουν να εκτελούνται εκ νέου.

Οι ρουτίνες συλλογικής επικοινωνίας πρέπει να περιλαμβάνουν όλες τις διεργασίες μέσα στο πεδίο του communicator. Εάν μία εργασία δε συμμετέχει στον communicator μπορεί να επηρεάσει το πρόγραμμα το οποίο θα παρουσιάσει μια ανεπιθύμητη συμπεριφορά συμπεριλαμβανομένης και την αποτυχία αυτού. Μέλημα του προγραμματιστή αποτελεί η διασφάλιση ότι όλες οι διεργασίες ενός communicator συμμετέχουν σε όλες τις συλλογικές λειτουργίες ανεξαιρέτως.

Το MPI κατέχει μια συγκεκριμένη λειτουργία που αφορά τις διεργασίες συγχρονισμού. Πρόκειται για την εντολή Comm.Barrier() που σχηματίζει ένα φράγμα και καμία διεργασία στον communicator δεν μπορεί να περάσει από το φράγμα μέχρι να καλέσουν όλοι την εντολή. Στα σχήματα που ακολουθούν φανταστείτε ότι ο οριζόντιος άξονας αναπαριστά την εκτέλεση του προγράμματος και τα κυκλάκια αναπαριστούν διαφορετικές διεργασίες.



Σχήμα 14: Φραγή (barrier)[18]

Ξεκινά πρώτα η διεργασία μηδέν που καλεί το Barrier στην T1 χρονική καταγραφή. Ενώ η διεργασία μηδέν βρίσκεται στο barrier, φθάνουν εκεί και οι διεργασίες ένα και τρία (χρονική καταγραφή T2). Όταν η διεργασία δύο καταλήγει και αυτή στο barrier (χρονική καταγραφή T3), όλες οι διεργασίες ξεκινούν εκ νέου εκτέλεση (χρονική καταγραφή T4).[18]

2.5.2 Καθολικές λειτουργίες επικοινωνίας

Στην MPI για την Python, οι μέθοδοι MPI.Comm.Bcast(), MPI.Comm.Scatter(), MPI.Comm.Gather(), MPI.Comm.Allgather(), MPI.Comm.Alltoall() και MPI.Comm.Alltoallw() παρέχουν υποστήριξη για τις συλλογικές επικοινωνίες των buffer μνήμης. Οι παραλλαγές με το μικρό πεζό γράμμα όπως MPI.Comm.bcast(), MPI.Comm.scatter(), MPI.Comm.gather(), MPI.Comm.allgather() και MPI.Comm.alltoall() μπορούν να επικοινωνούν με γενικά αντικείμενα της Python. Υποστηρίζονται επίσης οι vector παραλλαγές όπως MPI.Comm.Scatterv(), MPI.Comm.Gatherv(), MPI.Comm.Allgatherv(), MPI.Comm.Alltoallv() και MPI.Comm .Alltoallw() που μπορούν να μεταδίδουν διαφορετικές ποσότητες δεδομένων σε κάθε διεργασία.[14]

2.5.2.1 Εκπομπή(broadcast) Comm.Bcast(buf, root=0)

Παράμετροι

Comm (*MPI comm*) – επικοινωνητής μέσω του οποίου θα γίνει η εκπομπή

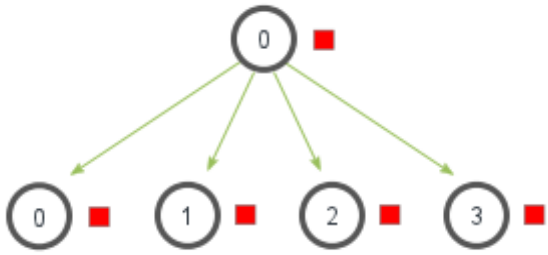
buf (*choice*) – buffer

root (*int*) – rank root λειτουργίας

Η εκπομπή ενισχύει τον παράλληλο προγραμματισμό δίνοντάς του μια δυναμική ιδιότητα, καθώς επιτρέπει στα δεδομένα που δημιουργούνται από τον κόμβο master να μπορούν να μεταδοθούν σε όλους τους υπόλοιπους κόμβους. Αυτό αποτρέπει την επαναλαμβανόμενη διαδικασία για την αποστολή δεδομένων σε κάθε κόμβο.

Παράδειγμα

```
1  from mpi4py import MPI
2  if rank == 0:
3      numbers = {'first':1,'second':2,'third':3}
4  else:
5          numbers = None
6  numbers = comm.bcast(numbers, root=0)
7  print  "rank", rank, numbers
```



Σχήμα 15: Broadcast λειτουργία [18]

Στο παραπάνω πρόγραμμα, στη μέθοδο bcast(), η πρώτη παράμετρος " numbers " αντιπροσωπεύει αυτό που πρέπει να μεταδοθεί και η δεύτερη παράμετρος "root = 0" υποδεικνύει από πού παίρνουμε τα δεδομένα. Εάν τρέξουμε αυτό το πρόγραμμα χρησιμοποιώντας 5 διεργασίες, το αποτέλεσμα πρέπει να μοιάζει με:

rank 0 {'first':1, 'second':2, 'third':3}

rank 4 {'first':1, 'second':2, 'third':3}

rank 3 {'first':1, 'second':2, 'third':3}

rank 1 {'first':1, 'second':2, 'third':3}

rank 2 {'first':1, 'second':2, 'third':3}

Μπορούμε να κάνουμε broadcast έναν numpy πίνακα χρησιμοποιώντας τη μέθοδο Bcast (παρατηρούμε το κεφάλαιο B). Εδώ θα τροποποιήσουμε τον κώδικα point-to-point για

να μεταδώσουμε τα δεδομένα πίνακα σε όλες τις διεργασίες στον επικοινωνητή(και όχι μόνο από τη διεργασία 0 στην 1):

```
1  from mpi4py import MPI
2  import numpy as np
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5
6  if rank == 0:
7      count = 5
8      data = np.linspace(0.0,20.0, count)      # δημιουργία πίνακα data στον rank 0
9  else:
10     count = None
11
12 # broadcast το count και διάθεση χώρου για τον πίνακα στους άλλους κόμβους
13 count = comm.bcast(count, root=0)
14 if rank != 0:
15     data = np.empty(count, dtype='d')
16
17 comm.Bcast(data, root=0) # broadcast τον data από rank 0 στους άλλους κόμβους
18 print('Rank: ',rank, ', received data: ',data)
```

Αυτός ο κώδικας παράγει την παρακάτω έξοδο:

('Rank: ', 0, ', received data: ', array([0., 5., 10., 15., 20.]))

('Rank: ', 1, ', received data: ', array([0., 5., 10., 15., 20.]))

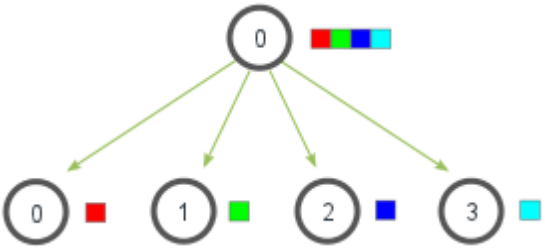
2.5.2.2 Διασπορά (scatter) Comm.Scatter(sendbuf, recvbuf, root)

Παράμετροι

sendbuf (choice) – διεύθυνση του buffer αποστολής (σημαντική μόνο στο root)

recvbuf (choice) – διεύθυνση του buffer λήψης

root (int) – rank της διεργασίας αποστολής



Σχήμα 16: Scatter λειτουργία[18]

Το scatter περιλαμβάνει μια καλά ορισμένη root διεργασία που στέλνει δεδομένα σε όλες τις υπόλοιπες διεργασίες σε έναν επικοινωνητή. Πρόκειται για μια συλλογική λειτουργία παρόμοια με το broadcast με μία όμως διαφορά αρκετά σημαντική. Συγκεκριμένα, το broadcast στέλνει τα ίδια δεδομένα σε όλες τις διεργασίες ενώ το scatter διασπά τον πίνακα των δεδομένων σε κομμάτια στέλνοντάς τα ξεχωριστά σε διαφορετικές διεργασίες.

Όπως βλέπουμε και στο παρακάτω σχήμα, η λειτουργία scatter διασπά έναν πίνακα από στοιχεία που στη συνέχεια τα διανέμει στις διεργασίες σύμφωνα με την τιμή του rank τους. Το πρώτο στοιχείο(με κόκκινο χρώμα) στέλνεται στη διεργασία μηδέν(0), το δεύτερο στοιχείο (με πράσινο χρώμα) στέλνεται στη διεργασία ένα(1), το τρίτο στοιχείο (με μπλε χρώμα) στη διεργασία δύο(2), και το τέταρτο στοιχείο (με γαλάζιο χρώμα) στη διεργασία τρία(3). Παρόλο που η root διεργασία (διεργασία 0) περιέχει ολόκληρο τον πίνακα των δεδομένων, η scatter λειτουργία θα αντιγράψει το κατάλληλο στοιχείο στο buffer λήψης της διεργασίας.

Η μέθοδος Comm.Scatter δέχεται τρία ορίσματα. Το πρώτο είναι ένας πίνακας δεδομένων που βρίσκεται στη root διεργασία, η δεύτερη παράμετρος χρησιμοποιείται για να κρατά τα ληφθέντα δεδομένα και η τελευταία παράμετρος υποδεικνύει τη root διεργασία που διασκορπίζει τον πίνακα των δεδομένων. Επίσης το μήκος του recvbuf πρέπει να διαιρεί εξίσου το μήκος του sendbuf, διαφορετικά χρησιμοποιούμε το Scatterv.[18]

Παράδειγμα

```
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
```

```
3  size=comm.Get_size()
4  rank=comm.Get_rank()
5  if rank == 0:
6      data = [(a+2) ** a for a in range (size)]
7      print 'Data before scatter', data
8  else:
9      data = None
10
11 data = comm.scatter(data, root=0)
12 print 'Rank', rank, 'has data: ', data
```

```
Data before scatter [1, 3, 16, 125]
Rank 0 has data:  1
Rank 1 has data:  3
Rank 2 has data:  16
Rank 3 has data:  125
```

Εδώ το μέγεθος των δεδομένων θα πρέπει να είναι ίσο με τον αριθμό των δεδομένων που αναμένεται, δηλαδή εάν τα δεδομένα έχουν 10 στοιχεία και υπάρχουν μόνο 5 διεργασίες τότε εμφανίζεται σφάλμα.

Ας δούμε κι εδώ ένα παράδειγμα με κεφαλαίο το S στο comm.scatter.

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  size=comm.Get_size()
6  rank=comm.Get_rank()
7
8  count = 5
9  data = None
10 if rank == 0:
```

```
11      data = np.linspace(1,size*count,count*size)
12      print ('Before scatter: ', data)
13
14  receiveBuffer = np.empty(count, dtype='d')
15  comm.Scatter(data, receiveBuffer, root=0)
16
17  print('Rank: ',rank, ', recvbuf received: ', receiveBuffer)
```

Σε αυτό το παράδειγμα, η διεργασία 0 δημιούργησε τα δεδομένα πίνακα. Στη συνέχεια τα δεδομένα αυτά διασκορπίζονται σε όλες τις διεργασίες (συμπεριλαμβανομένης της διεργασίας 0) χρησιμοποιώντας την εντολή comm.Scatter. Να σημειωθεί όμως ότι αρχικά έπρεπε να αρχικοποιήσουμε (να δημιουργήσουμε) τον buffer πίνακα receiveBuffer που θα λάβει αυτά τα δεδομένα.

Η έξοδος αυτού του κώδικα είναι η παρακάτω:

```
('Before scatter: ', array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.]))
('Rank: ', 0, ', recvbuf received: ', array([1., 2., 3., 4., 5.]))
('Rank: ', 1, ', recvbuf received: ', array([ 6.,  7.,  8.,  9., 10.]))
```

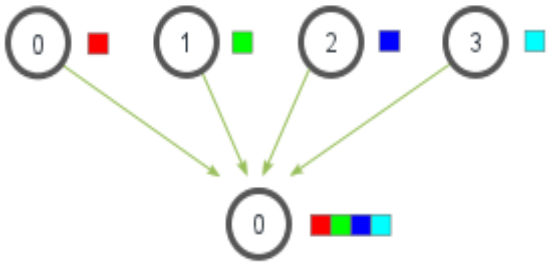
2.5.2.3 Συγκέντρωση (gather) Comm.Gather(sendbuf, recvbuf, root)

Παράμετροι

sendbuf (choice) – διεύθυνση του buffer αποστολής

recvbuf (choice) – διεύθυνση του buffer λήψης

root (int) – rank της διεργασίας παραλαβής



Σχήμα 17: Gather λειτουργία[18]

Η συλλογή(gather) είναι το αντίστροφο της διασποράς (scatter). Αντί της διασποράς των στοιχείων από μια διεργασία σε πολλές διεργασίες, η συλλογή παίρνει στοιχεία από πολλές διεργασίες και τα συγκεντρώνει σε μία ενιαία διεργασία. Αυτή η ρουτίνα είναι

ιδιαίτερα χρήσιμη σε πολλούς παράλληλους αλγορίθμους, όπως η παράλληλη ταξινόμηση και αναζήτηση.

Παρόμοια με το scatter, το gather παίρνει στοιχεία από κάθε διεργασία και τα συγκεντρώνει στη root διεργασία. Τα στοιχεία ταξινομούνται από το rank της διεργασίας από την οποία ελήφθησαν. Η μέθοδος Comm.Gather παίρνει τα ίδια ορίσματα με το Comm.Scatter. Ωστόσο, στη gather λειτουργία, μόνο η root διεργασία πρέπει να έχει ένα έγκυρο buffer λήψης.[18]

Παράδειγμα: (Πρόκειται για το ίδιο παράδειγμα με το προηγούμενο για το scatter αλλά με κάποιες αλλαγές)

```
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
3  size=comm.Get_size()
4  rank=comm.Get_rank()
5
6   if rank == 0:
7       data = [(a+1) ** a for a in range (size)]
8       print 'Before scatter', data
9   else:
10     data = None
11
12  data = comm.scatter(data,root=0)
13  print 'Rank',rank,'has data: ', data
14
15  collection= comm.gather(data, root=0)
16  if rank == 0:
17  print  'Rank 0 gathered data: ', collection
```

Η έξοδος για αυτό το πρόγραμμα θα είναι:

Before scatter [1, 2, 9, 64]

Rank 0 has data: 1

```
Rank 1 has data:  2
Rank 2 has data:  9
Rank 3 has data:  64
Rank 0 gathered data:  [1, 2, 9, 64]
```

Στην παραπάνω έξοδο, η τελευταία γραμμή είναι λόγω της gather. Ένα άλλο πράγμα που μπορεί να γίνει είναι να αλλάξουν τα δεδομένα μόλις διασκορπιστούν, δηλαδή όταν κάθε κόμβος λαμβάνει μέρος των scattered δεδομένων. Παράδειγμα: data= data*2

Έτσι, όταν συμβαίνει η συλλογή, τα δεδομένα δεν είναι ίδια με αυτά που στάλθηκαν αλλά διατηρούν το αλλαγμένο πρότυπο αποδεικνύοντας έτσι ότι λειτουργεί σωστά.

Ας δούμε ένα ακόμη παράδειγμα με χρήση όμως της comm.Gather (κεφαλαίο το G).

```
1  from mpi4py import MPI
2  import numpy as np
4  comm = MPI.COMM_WORLD
5  size=comm.Get_size()
6  rank=comm.Get_rank()
7
8  count = 5
9  sendBuffer = np.linspace(rank*count+1,(rank+1)*count,count)
10 print('Rank: ',rank, ', has data: ',sendBuffer)
11
12 receiveBuffer = None
13 if rank == 0:
14     receiveBuffer = np.empty(count*size, dtype='d')
15
16 comm.Gather(sendBuffer, receiveBuffer, root=0)
17
18 if rank == 0:
19         print('Rank: ',rank, 'gathered data: ',receiveBuffer)
```

Η έξοδος αυτού του κώδικα είναι η παρακάτω:

```
('Rank: ', 0, ', has data: ', array([1., 2., 3., 4., 5.]))
('Rank: ', 1, ', has data: ', array([ 6.,  7.,  8.,  9., 10.]))
```

('Rank: ', 0, 'gathered data: ', array([1., 2., 3., 4., 5., 6., 7., 8., 9., 10.]))

2.5.3 Καθολικές λειτουργίες μείωσης(reduction)

Η reduce λειτουργία του MPI παίρνει τιμές από έναν πίνακα κάθε διεργασίας και τις μαζεύει (reduces) σε ένα μόνο αποτέλεσμα στη root διεργασία. Αυτό είναι ουσιαστικά σαν να έχουμε μια πολύ περίπλοκη εντολή αποστολής από κάθε διεργασία στη root διεργασία και, στη συνέχεια, έχουμε τη root διεργασία να εκτελεί τη reduction λειτουργία. Ευτυχώς, το MPI κάνει reduce όλα αυτά με μία συνοπτική εντολή. Οι λειτουργίες καθολικής(μείωσης) στα buffer μνήμης είναι προσβάσιμες μέσω των μεθόδων MPI.Comm.Reduce(), MPI.Comm.Reduce_scatter(), MPI.Comm.Allreduce(), MPI.Intracomm.Scan() και MPI.Intracomm.Exscan(). Οι παραλλαγές με το μικρό πεζό γράμμα όπως MPI.Comm.reduce(), MPI.Comm.allreduce(), MPI.Intracomm.scan() και MPI.Intracomm.exscan() μπορούν να μεταδίδουν γενικά αντικείμενα της Python. Ωστόσο, οι πραγματικοί απαιτούμενοι υπολογισμοί μείωσης πραγματοποιούνται διαδοχικά σε κάποια διεργασία. Μπορούν να εφαρμοστούν όλες οι προκαθορισμένες διεργασίες μείωσης (δηλ. MPI.SUM, MPI.PROD, MPI.MAX, κλπ.).[14]

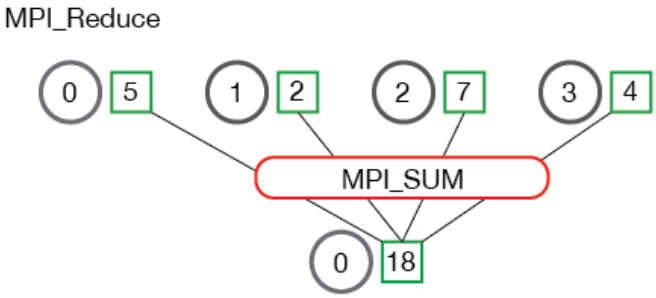
- **<u>Comm.Reduce(sendbuf, recvbuf, Op op = MPI.SUM, root = 0)</u>**

  Reduces τις τιμές σε όλες τις διεργασίες σε μία μόνο τιμή στη root διεργασία.

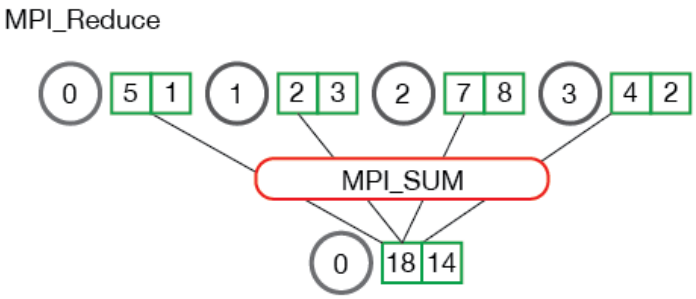
  <u>Παράμετροι</u>

  Comm (MPI comm) – επικοινωνητής που θέλουμε να υποβάλλουμε ερώτημα
  sendbuf (choice) – διεύθυνση του buffer αποστολής
  recvbuf (choice) – διεύθυνση του buffer λήψης (σημαντική μόνο στο root)
  op (handle) – reduce λειτουργία
  root (int) – rank της root λειτουργίας

Στα διπλανό σχήμα, κάθε διεργασία περιέχει έναν ακέραιο αριθμό. Η διεργασία 0 καλεί τη reduction διεργασία που χρησιμοποιεί ως reduction λειτουργία το



MPI_SUM. Οι τέσσερις αριθμοί αθροίζονται στο αποτέλεσμα το οποίο αποθηκεύεται στη root διεργασία.
Είναι επίσης χρήσιμο να δούμε τι συμβαίνει όταν οι διεργασίες περιέχουν πολλά στοιχεία.



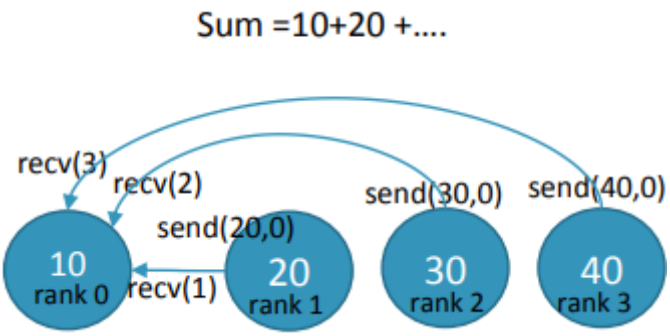
Σχήμα 18: Reduction λειτουργία[18]

Στο παραπάνω σχήμα φαίνεται η reduction λειτουργία πολλαπλών αριθμών ανά διεργασία.

Ας δούμε τώρα ένα παράδειγμα με send και recv και για το πώς η χρήση του reduce θα αλλάξει τον κώδικα προς το καλύτερο.

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank=comm.Get_rank()
size=comm.Get_size()

num = (rank+1)*10
print "Rank %d has number %d" %(rank, num)
if rank == 0:
    sum = num
    for i in range(1,size):
        sum += comm.recv(source=i)
     print "Rank 0 has sum of nums %d" %sum
else:
    comm.send(num, dest=0)
```



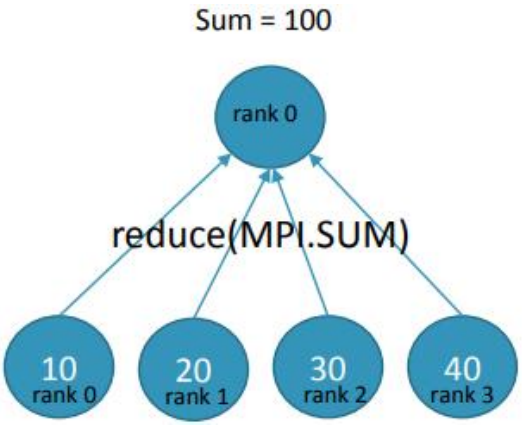
Σχήμα 19 : Υπολογισμός αθροίσματος στο Rank 0 [20]

Κάθε διεργασία στέλνει μια τιμή στο rank 0 τις οποίες τις συγκεντρώνει στο τέλος και υπολογίζει το άθροισμα. Το rank 0 δε χρειάζεται να στείλει στον εαυτό του.
Η έξοδος αυτού του κώδικα είναι η παρακάτω:
Rank 0 has number 10
Rank 1 has number 20
Rank 0 has sum of nums 30

Να σημειωθεί ότι το rank 0 "λαμβάνει" από το rank 1, το rank 2 και το rank 3 κατά σειρά. Κάθε διεργασία αρχίζει να "στέλνει" μόλις αρχίσει να εκτελείται, αλλά το "send" ολοκληρώνεται μόνο όταν το αντίστοιχο "Recv" καλείται από το Rank 0.

Έχοντας αυτή τη "σειριακή" ρουτίνα σε παράλληλο κώδικα δεν είναι και το ιδανικό σενάριο. Με μόνο 4 διεργασίες, αυτό μπορεί να μην ακούγεται σαν μια μεγάλη υπόθεση, αλλά αυτό μπορεί να είναι πολύ αναποτελεσματικό όταν έχουμε, ας πούμε, 1000 διεργασίες. Οι τιμές που αποστέλλονται σειριακά διακυβεύουν το σκοπό και το λόγο ύπαρξης του παράλληλου προγραμματισμού.

Ας δούμε τώρα τον ακόλουθο κώδικα:



Σχήμα 20: Υπολογισμός αθροίσματος στον Rank 0 με χρήση του Reduce[20]

```
1  from mpi4py import MPI
2  comm = MPI.COMM_WORLD
3  rank = comm.Get_rank()
4  size = comm.Get_size()
5
6  num = (rank+1)*10
7  print "Rank %d has number %d" %(rank, num)
8
9  sum = comm.reduce(num, op=MPI.SUM, root=0)
10
11 if rank==0:
12         print "Rank 0 has sum of nums %d" %sum
```

Η έξοδος αυτού του κώδικα είναι φυσικά η ίδια όπως και παραπάνω:

Rank 0 has number 10

Rank 1 has number 20

Rank 0 has sum of nums 30

Αυτό το πρόγραμμα παράγει το ίδιο αποτέλεσμα, αλλά χρησιμοποιεί μια συλλογική κλήση, τη "reduce". Αυτή η συνάρτηση επιφέρει την αποστολή της τιμής του " num " από

κάθε διεργασία στη root διεργασία (στην προκειμένη περίπτωση Rank 0) και εφαρμόζει τη λειτουργία του αθροίσματος σε όλες τις τιμές. Ως αποτέλεσμα, οι πολλαπλές τιμές να μειώνονται (become reduced) σε μία μόνο τιμή.

Παρακάτω βλέπουμε ένα άλλο παράδειγμα με χρήση όμως της comm.Reduce(R κεφαλαίο).

Παράδειγμα

```
1  from mpi4py import MPI
2  import numpy as np
3  comm = MPI.COMM_WORLD
4  size=comm.Get_size()
5  rank=comm.Get_rank()
6
7  number = np.array(rank+2,dtype=int)
8  print('Rank: ',rank, 'has number = ', number)
9
10 total = np.array(0,dtype=int)
11 maxValue = np.array(0,dtype=int)
12 minValue = np.array(0,dtype=int)
13
14 comm.Reduce(number, total, op=MPI.SUM, root=0)
15 comm.Reduce(number, maxValue, op=MPI.MAX, root=0)
16 comm.Reduce(number, minValue, op=MPI.MIN, root=0)
17
18 if rank == 0:
19     print(' Rank 0: found sum =',total)
20     print(' Rank 0: found max =',maxValue)
21     print(' Rank 0: found min =',minValue)
```

Η έξοδος αυτού του κώδικα έχει ως εξής:

('Rank: ', 0, 'has number = ', array(2))

('Rank: ', 1, 'has number = ', array(3))

(' Rank 0: found sum =', array(5))

(' Rank 0: found max =', array(3))

(' Rank 0: found min =', array(2))

- **Comm.Allreduce(sendbuf, recvbuf, Op op = MPI.SUM)**

  Reduces τις τιμές σε όλες τις διεργασίες σε μία μόνο τιμή σε όλες τις διεργασίες.

  Παράμετροι

  Comm (MPI comm) – επικοινωνητής που θέλουμε να υποβάλλουμε ερώτημα
  sendbuf (choice) – αρχική διεύθυνση του buffer αποστολής
  recvbuf (choice) – αρχική διεύθυνση του buffer λήψης
  op (handle) – λειτουργία

Παράδειγμα

Ίδιο με το παράδειγμα για το Reduce, με αντικατάσταση όμως του comm.Reduce με το comm.Allreduce. Πλέον όλες οι διεργασίες έχουν τώρα την "reduced" τιμή.

Να σημειωθεί ότι εντός της κλάσης Op υπάρχουν τιμές που αντιπροσωπεύουν προκαθορισμένες λειτουργίες που θα χρησιμοποιηθούν με τις "reduce" λειτουργίες. Υπάρχουν επίσης μέθοδοι που επιτρέπουν τη δημιουργία λειτουργιών που ορίζονται από το χρήστη.

2.5.3.1 Η κλάση Op (Reduction χειριστές)

Αυτές είναι οι προκαθορισμένες λειτουργίες που μπορούν να χρησιμοποιηθούν για τις παραμέτρους εισόδου Op (χρήση: comm.Reduce (sendbuf, recvbuf, op = MPI.PROD)).

Name	Meaning
MPI.MAX	μέγιστο
MPI.MIN	ελάχιστο
MPI.SUM	άθροισμα
MPI.PROD	product
MPI.LAND	λογικό και
MPI.BAND	bit-wise και
MPI.LOR	λογικό ή
MPI.BOR	bit-wise ή
MPI.LXOR	λογικό xor
MPI.BXOR	bit-wise xor
MPI.MAXLOC	μέγιστο και location
MPI.MINLOC	ελάχιστο και location

Πίνακας 5 : Προκαθορισμένες λειτουργίες κλάσης Op[12]

Η επικοινωνία μεταξύ των διεργασιών είναι δαπανηρή. Επειδή κάθε μήνυμα πρέπει να αποστέλλεται μέσω κάποιου δικτύου, πρέπει να ελαχιστοποιήσουμε και να βελτιστοποιήσουμε αυτή την επικοινωνία μεταξύ των διεργασιών.

Σημείωση: Υπάρχουν δύο βασικές αρχές που πρέπει να θυμόμαστε:

- **Εξισορρόπηση φορτίου:** Ένα πρόγραμμα κρίνεται ως μη αποτελεσματικό εάν δεν χρησιμοποιεί όλους τους διαθέσιμους πόρους (π.χ. αν οι διεργασίες βρίσκονται σε αδράνεια επειδή η μία περιμένει την άλλη)
- **Κόστος επικοινωνίας:** Ο λόγος ύπαρξης των Broadcast και Reduce είναι για τη βελτιστοποίηση της όποιας μορφής επικοινωνίας μεταξύ του επικοινωνητή. Εντούτοις όμως, κάθε μορφή μεταβίβασης δεδομένων ή μηνυμάτων κοστίζει εξαιρετικά και αποτελεί ένα από τα κύρια εμπόδια για την επίτευξη επιτάχυνσης όταν παραλληλίζεται ένας αλγόριθμος.[12]

2.6 Εμβάθυνση στις έννοιες Broadcast(Εκπομπή) και Reduce(Μείωση)

Ας δούμε πιο προσεκτικά την κλήση στο comm.Bcast. Το Bcast στέλνει δεδομένα από μια διεργασία σε όλες τις άλλες. Το πρώτο της όρισμα είναι ένας πίνακας δεδομένων, που πρέπει να υπάρχει σε όλους τους επεξεργαστές. Ωστόσο, αυτό που περιέχεται σε αυτόν τον πίνακα μπορεί να είναι ασήμαντο. Το δεύτερο όρισμα λέει στο Bcast ποια διεργασία έχει τις χρήσιμες πληροφορίες. Στη συνέχεια προχωρεί να αντικαταστήσει οποιαδήποτε δεδομένα στους πίνακες όλων των άλλων διεργασιών.

Το Bcast συμπεριφέρεται σαν να είναι σύγχρονο. "Σύγχρονο" σημαίνει ότι όλες οι διεργασίες είναι σε συγχρονισμό μεταξύ τους, σαν να ελέγχονται από ένα γενικό ρολόι. Για παράδειγμα, εάν όλες οι διεργασίες κάνουν μια κλήση στο Bcast, είναι εγγυημένο ότι καλούν την υπορουτίνα ουσιαστικά την ίδια στιγμή.

Για πρακτικούς λόγους, μπορούμε επίσης να σκεφτούμε και το Reduce ως σύγχρονο. Η διαφορά είναι ότι το Reduce έχει μόνο μία διεργασία λήψης και αυτή η διεργασία είναι η μόνη διεργασία της οποίας τα δεδομένα είναι εγγυημένο ότι περιέχουν τη σωστή τιμή κατά την ολοκλήρωση της κλήσης. Για παράδειγμα, ας υποθέσουμε ότι προσθέτουμε τον αριθμό 1 καλώντας το Comm.Reduce σε 100 διεργασίες, με το root να είναι η διεργασία 0. Η τεκμηρίωση του Comm.Reduce μας λέει ότι το buffer λήψης της διεργασίας 0 θα περιέχει τον αριθμό 100. Τι θα υπάρχει στο buffer λήψης της διεργασίας 1;

Η απάντηση είναι ότι δεν ξέρουμε. Το Reduce θα μπορούσε να εφαρμοστεί με ποικίλους τρόπους και με διάφορους παράγοντες, και ως εκ τούτου δεν είναι ντετερμινιστικό. Κατά τη συλλογική επικοινωνία, όπως το Reduce, δεν έχουμε καμία εγγύηση για την τιμή που θα βρίσκεται στην προσωρινή μνήμη λήψης των ενδιάμεσων διεργασιών. Το πιο σημαντικό είναι ότι οι μελλοντικοί υπολογισμοί δεν θα πρέπει ποτέ να βασίζονται σε αυτά τα δεδομένα εκτός από το root της διεργασίας (διεργασία 0 στην περίπτωσή μας).[12]

2.7 Δυναμική Διαχείριση Διεργασιών

Σύμφωνα με την προδιαγραφή MPI-1, μια παράλληλη εφαρμογή θεωρούνταν στάσιμη, με άλλα λόγια, δεν μπορούσαμε να ενσωματώσουμε ή να αφαιρέσουμε διεργασίες από μια εφαρμογή αν αυτή είχε ξεκινήσει να εκτελείται. Ευτυχώς όμως αυτή η αδυναμία

συζητήθηκε στο MPI-2. Η νέα αυτή προδιαγραφή ενισχύθηκε με ένα μοντέλο διαχείρισης διεργασιών που διαθέτει μια ουσιαστική διεπαφή ανάμεσα σε μια εφαρμογή και στους εξωτερικούς πόρους καθώς και στους διαχειριστές των διεργασιών.

Αυτή η επέκταση MPI-2 μπορεί να θεωρηθεί πραγματικά χρήσιμη, συγκεκριμένα για σειριακές εφαρμογές που βρίσκονται σε παράλληλα modules ή για παράλληλες εφαρμογές με μοντέλο πελάτη / διακομιστή. Η προδιαγραφή MPI-2 διαθέτει έναν μηχανισμό που βοηθά στη δημιουργία νέων διεργασιών καθώς και στην επικοινωνία μεταξύ τους και της υπάρχουσας εφαρμογής MPI. Διαθέτει επίσης μηχανισμούς για την καθιέρωση της επικοινωνίας μεταξύ δύο εφαρμογών MPI, ακόμη και όταν η μία δεν ξεκίνησε την άλλη.

Στην MPI για την Python, σε περίπτωση που καλέσουμε τη μέθοδο MPI.Intracomm.Spawn() μέσα σε έναν intra-communicator, υπάρχει η δυνατότητα δημιουργίας νέων αυτόνομων ομάδων διεργασιών. Η κλήση αυτή επιστρέφει έναν νέο inter-communicator (που αποτελεί ένα στιγμιότυπο MPI.Intercomm) στην ομάδα διεργασίας - γονέα. Από την άλλη πλευρά, η ομάδα διεργασίας - παιδί μπορεί να ανακτήσει τον αντίστοιχο inter-communicator καλώντας τη μέθοδο κλάσης MPI.Comm.Get_parent(). Ο νέος inter-communicator μπορεί να χρησιμοποιηθεί προκειμένου να επιτευχθεί επικοινωνία από σημείο σε σημείο και συλλογική ανάμεσα στις ομάδες διεργασιών γονέα και παιδιού.

Εναλλακτικά, διαφορετικές ομάδες διεργασιών μπορούν να εγκαθιδρύσουν επικοινωνία χρησιμοποιώντας μια προσέγγιση πελάτη / διακομιστή. Κάθε εφαρμογή διακομιστή αρχικά πρέπει να καλέσει τη συνάρτηση MPI.Open_port() προκειμένου να ανοίξει μια θύρα επικοινωνίας, τη συνάρτηση MPI.Publish_name() για να δημοσιεύσει μια παρεχόμενη υπηρεσία και στη συνέχεια να καλέσει τη μέθοδο MPI.Intracomm.Accept(). Οι εφαρμογές πελάτη έχουν τη δυνατότητα να βρουν μια δημοσιευμένη υπηρεσία καλώντας τη συνάρτηση MPI.Lookup_name(), η οποία επιστρέφει τη θύρα επικοινωνίας με ένα διακομιστή και στη συνέχεια να καλέσουν τη συνάρτηση MPI.Intracomm.Connect(). Και οι δύο συναρτήσεις MPI.Intracomm.Accept() και MPI.Intracomm.Connect() επιστρέφουν ένα στιγμιότυπο MPI.Intercomm. Όταν πλέον δε χρειάζεται η σύνδεση ανάμεσα στις διεργασίες πελάτη / διακομιστή, τότε όλοι πρέπει να καλέσουν τη μέθοδο MPI.Comm.Disconnect(). Επιπλέον, οι εφαρμογές διακομιστή θα πρέπει να απελευθερώνουν πόρους καλώντας τις λειτουργίες MPI.Unpublish_name() και MPI.Close_port(). [14]

2.8 Μονομερείς επικοινωνίες

Οι μονομερείς επικοινωνίες (που ονομάζεται επίσης Απομακρυσμένη πρόσβαση μνήμης, RMA) συμπληρώνουν το γνωστό αμφίδρομο μοντέλο επικοινωνίας MPI αποστολής / λήψης, με ένα μονομερές put / get περιβάλλον. Η μονομερής επικοινωνία μπορεί να αξιοποιήσει τις δυνατότητες του εξειδικευμένου υλικού του δικτύου. Επιπρόσθετα, αυτή η προσθήκη ελαττώνει την καθυστέρηση και το overhead του λογισμικού χρησιμοποιώντας ένα παράδειγμα διαμοιραζόμενης μνήμης.

Η MPI χρησιμοποιεί αντικείμενα που ονομάζονται παράθυρα. Καθορίζουν διαισθητικά περιοχές της μνήμης μιας διεργασίας που έχουν διατεθεί για λειτουργίες απομακρυσμένης ανάγνωσης και εγγραφής. Στα μπλοκ μνήμης μπορεί κανείς να αποκτήσει πρόσβαση μέσω τριών λειτουργιών για put (απομακρυσμένη αποστολή), get (απομακρυσμένη εγγραφή) και accumulate (απομακρυσμένη ενημέρωση ή μείωση) στοιχείων δεδομένων. Ένας πολύ μεγαλύτερος αριθμός λειτουργιών υποστηρίζει διαφορετικά στυλ συγχρονισμού η σημασιολογία των οποίων είναι αρκετά περίπλοκη.

Στην MPI για την Python, μπορεί κανείς να χρησιμοποιήσει τις μονομερείς λειτουργίες μέσω στιγμιοτύπων της κλάσης MPI.Win. Καλώντας τη μέθοδο MPI.Win.Create() στο πλήθος των διεργασιών, μπορούμε να δημιουργήσουμε νέα αντικείμενα παραθύρου μέσα σε έναν επικοινωνητή και όταν πλέον δε χρειάζεται ένα στιγμιότυπο παραθύρου θα πρέπει να καλούμε τη μέθοδο MPI.Win.Free().[14]

2.9 Παράλληλη Είσοδος / Έξοδος

Το πρότυπο POSIX παρέχει ένα μοντέλο φορητού συστήματος αρχείων. Ωστόσο, η βελτιστοποίηση που απαιτείται για την παράλληλη είσοδο / έξοδο δεν μπορεί να επιτευχθεί με αυτή τη γενική διεπαφή. Κρίθηκε λοιπόν αναγκαίο το παράλληλο σύστημα εισόδου / εξόδου να μπορεί να παρέχει μια διασύνδεση υψηλού επιπέδου που θα υποστηρίζει την κατανομή των δεδομένων αρχείων μεταξύ των διεργασιών και μια συλλογική διεπαφή που

θα υποστηρίζει την πλήρη μεταφορά των καθολικών δομών των δεδομένων μεταξύ των μνημών διεργασίας και των αρχείων. Ενσωματώθηκε λοιπόν στο πρότυπο MPI-2 μια προσαρμοσμένη διεπαφή που θα υποστήριζε πιο σύνθετες λειτουργίες παράλληλης εισόδου / εξόδου.

Η προδιαγραφή MPI για παράλληλη είσοδο / έξοδο χρησιμοποιεί αντικείμενα που ονομάζονται αρχεία. Όπως ορίζεται από το MPI, τα αρχεία δεν είναι απλώς συνεχείς ροές από bytes αλλά θεωρούνται καθορισμένες συλλογές στοιχείων δεδομένων. Το MPI υποστηρίζει σειριακή ή τυχαία πρόσβαση σε οποιοδήποτε ολοκληρωμένο σύνολο αυτών των στοιχείων.

Τα κοινά πρότυπα για να έχει κανείς πρόσβαση σε ένα διαμοιραζόμενο αρχείο (εκπομπή, διασκορπισμός, συλλογή, μείωση) εκφράζονται χρησιμοποιώντας τύπους δεδομένων που ορίζονται από το χρήστη. Η δυνατότητα αυτή έχει το πλεονέκτημα της προστιθέμενης ευελιξίας και εκφραστικότητας.

Στην MPI για την Python, όλες οι MPI λειτουργίες εισόδου κι εξόδου μπορούν να εκτελεστούν με τη χρήση στιγμιοτύπων της κλάσης MPI.File. Για να τα διαχειριστούμε καλούμε τη μέθοδο MPI.File.Open() σε όλες τις διεργασίες ενός Communicator δίνοντας ένα όνομα αρχείου και τον επιθυμητό τρόπο πρόσβασης. Μετά τη χρήση, πρέπει να κλείσουν καλώντας τη μέθοδο MPI.File.Close() ή ακόμα και να διαγραφούν με τη κλήση της συνάρτησης MPI.File.Delete().

Μετά τη δημιουργία, τα αρχεία συσχετίζονται συνήθως με μια προβολή ανά διεργασία που μπορούμε να την ορίσουμε και να της θέσουμε ερωτήματα με χρήση των μεθόδων MPI.File.Set_view() και MPI.File.Get_view() αντίστοιχα.

Οι πραγματικές λειτουργίες εισόδου / εξόδου επιτυγχάνονται με πολλές μεθόδους που συνδυάζουν κλήσεις ανάγνωσης και εγγραφής με διαφορετική συμπεριφορά όσον αφορά την τοποθέτηση, το συντονισμό και τον συγχρονισμό. Ολοκληρώνοντας, το MPI για την Python παρέχει τριάντα (30) μεθόδους που ορίζονται στο MPI-2 για ανάγνωση ή εγγραφή σε αρχεία χρησιμοποιώντας ρητά offsets ή δείκτες αρχείου (μεμονωμένους ή κοινόχρηστους), σε παρεμποδιστικές ή μη παρεμποδιστικές και συλλογικές ή μη συλλογικές εκδόσεις.[14]

Κεφάλαιο 3

Συγκριτική μελέτη απόδοσης C με Python

Για τα προγράμματα που ακολουθούν χρησιμοποιήθηκαν τα παρακάτω:

- Λειτουργικό σύστημα Ubuntu 18.04.3 LTS
- Gcc compiler 7.4.0
- Python v. 2.7.17
- Πακέτο mpi4py μέσω της εντολής: **$ [sudo] pip install mpi4py**

Για τις γραφικές παραστάσεις χρησιμοποιήθηκε το πρόγραμμα gnuplot v. 5.2 patchlevel 7.

3.1 Πρώτοι αριθμοί

Σύμφωνα με την Βικιπαίδεια [24], στα μαθηματικά πρώτος αριθμός (ή απλά πρώτος) είναι ένας φυσικός αριθμός μεγαλύτερος της μονάδας με την ιδιότητα οι μόνοι φυσικοί διαιρέτες του να είναι η μονάδα και ο εαυτός του. Η σειριακή έκδοση του προγράμματος είναι η ακόλουθη:

```
1  for (i=1; i<=N; i++) {
2        prime = 0;
3        for (d=2; d<=i/2; d++)
4          if (i % d == 0)
5               prime = 1;
6        if (prime == 0)
7             total++;
8  }
```

Η αντίστοιχη μορφή του κατανεμημένου προγράμματος με την C είναι:

```
1  for (i=rank+1; i<=N; i+=size) {
2        prime = 0;
3        for (d=2; d<=i/2; d++)
4          if (i % d == 0)
5               prime = 1;
6        if (prime == 0)
7              subtotal ++;
8  }
```

Ενώ η αντίστοιχη μορφή με την Python είναι η ακόλουθη:

```
1  for i in range (rank+1,N+1,nprocs):
2        prime = 0;
3        for d in range(2,(i/2)+1):
4          if (i % d == 0)
5               prime = 1;
6        if (prime == 0)
7              subtotal=subtotal+1
8  }
```

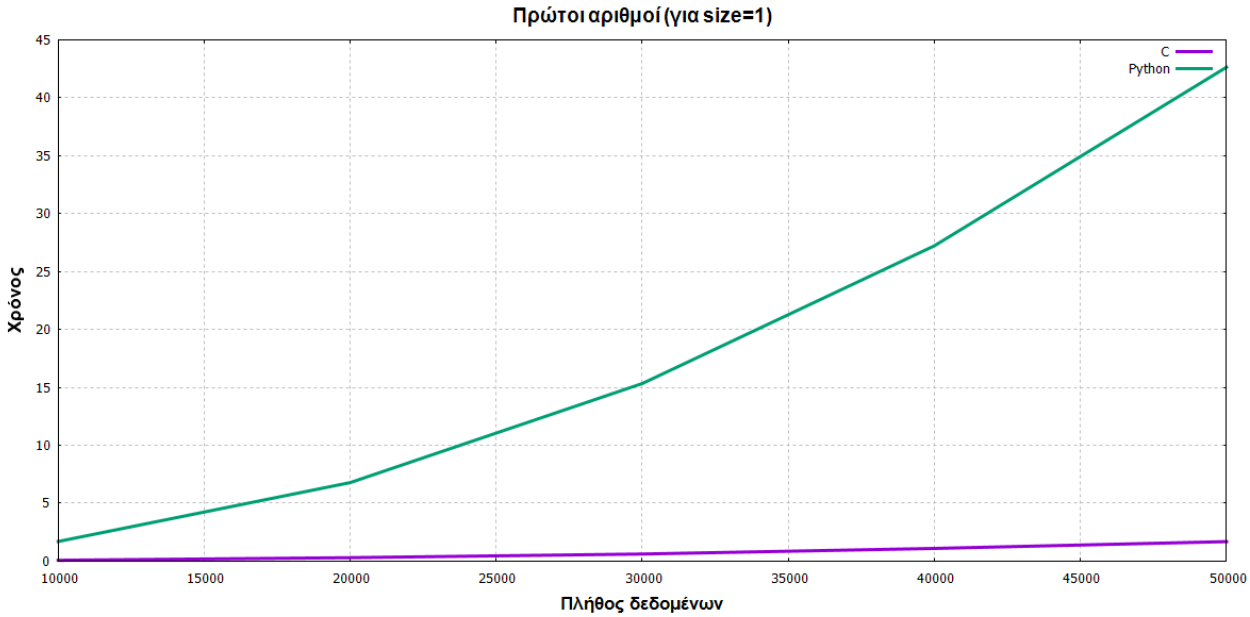
Τα παραπάνω προγράμματα εκτελέστηκαν για όγκο δεδομένων N=10000, 20000, 30000, 40000 και 50000 και χρησιμοποιήθηκαν για το σκοπό αυτό size = 1, 2, και 4 επεξεργαστές.

Προκύπτει έτσι ο παρακάτω πίνακας:

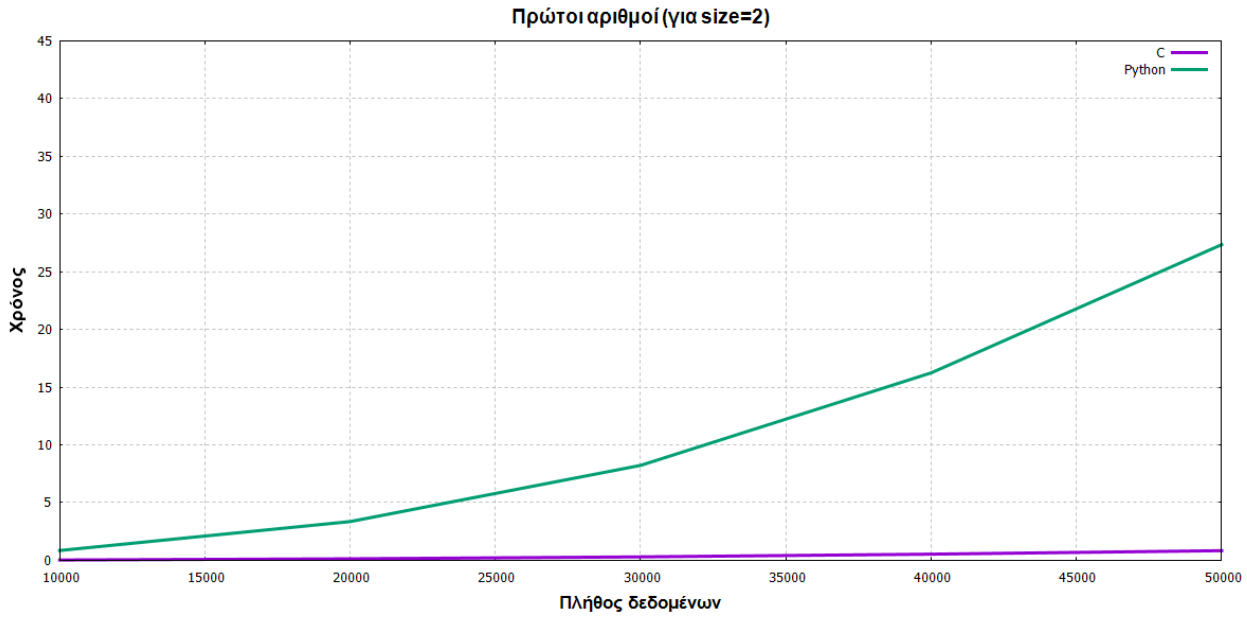
N/size	Size=1		Size=2		Size=4	
	C	Python	C	Python	C	Python
10000	0.0836043357849121	1.7043957710266113	0.0352699756622314	0.8572199344635010	0.0176150798797607	0.8118720054626465
20000	0.3035266399383545	6.7881028652191162	0.1416757106781006	3.3644821643829346	0.0696809291839600	2.7373080253601074
30000	0.6228024959564209	15.3419950008392334	0.3036546707153320	8.2341680526733398	0.1528496742248535	5.2792599201202393
40000	1.1041328907012939	27.2287440299987793	0.5369086265563965	16.2450649738311768	0.2786495685577393	9.8717911243438721
50000	1.6878249645233154	42.6424059867858887	0.8405830860137939	27.3530459403991699	0.4201958179473877	16.1333301067352295

Ακολουθούν οι γραφικές παραστάσεις για size = 1, 2 και 4.

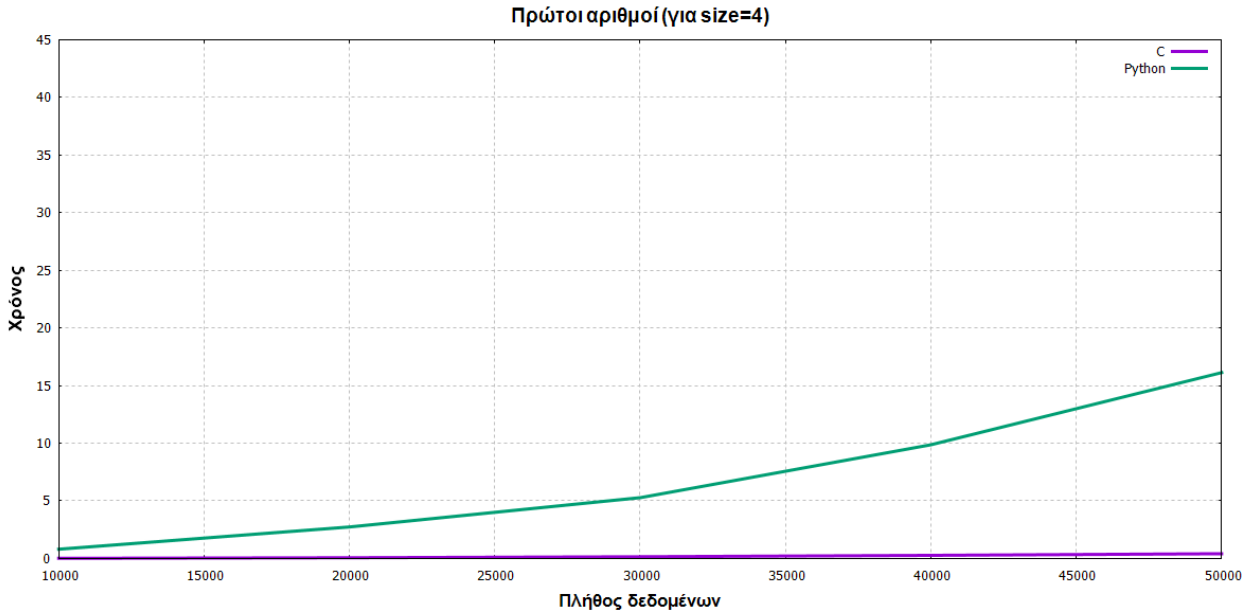
- **Για size = 1:**



- **Για size = 2:**



- **Για size = 4:**



3.2 Υπολογισμός του π=3,14

Ο αριθμός π είναι μια μαθηματική σταθερά οριζόμενη ως ο λόγος της περιφέρειας προς τη διάμετρο ενός κύκλου. Θεωρείται ένας άρρητος αριθμός, κάτι που σημαίνει ότι δεν μπορεί να εκφραστεί ακριβώς ως λόγος δύο ακεραίων. Μάλιστα θεωρείται υπερβατικός αριθμός, δηλαδή δεν αποτελεί ρίζα ενός μη-μηδενικού πολυωνύμου με ρητούς συντελεστές.

Οι μαθηματικοί δεν ασχολούνται πλέον μιας και είναι αδύνατο να υπολογιστεί επακριβώς. Οι προγραμματιστές όμως παρόλα αυτά συνεχίζουν την προσπάθεια γιατί θεωρείται ένα από τα καλύτερα κριτήρια για να ελεγχθεί η ισχύ ενός αλγορίθμου.[25]

Το σειριακό πρόγραμμα υπολογισμού του π φαίνεται παρακάτω:

```
1  h = 1.0 / (double) n;
2  sum = 0.0;
3  for (i=1;  i<=n;  i++) {
4          x = h * ((double) i - 0.5);
5          sum += (4.0 / (1 + x * x));
6  }
7  mypi = h * sum;
```

Η αντίστοιχη μορφή του κατανεμημένου προγράμματος με την C είναι:

```
1  h = 1.0 / (double) n;
2  sum = 0.0;
3       for (i=rank+1; i<=n; i+=size) {
4          x = h * ((double) i - 0.5);
5          sum += (4.0 / (1 + x * x));
6  }
7  mypi = h * sum;
```

Ενώ η αντίστοιχη μορφή με την Python είναι η ακόλουθη:

```
1  dx = 1.0 / nsteps
2  partial_pi = 0.0
3  for i in range(data[0], data[1]):
4          x = (i + 0.5) * dx
5          partial_pi += 4.0 / (1.0 + x * x)
6  partial_pi *= dx
```

όπου data είναι ένας πίνακας στον οποίο αποθηκεύονται ο πρώτος και ο τελευταίος δείκτης από κάθε υποεργασία.

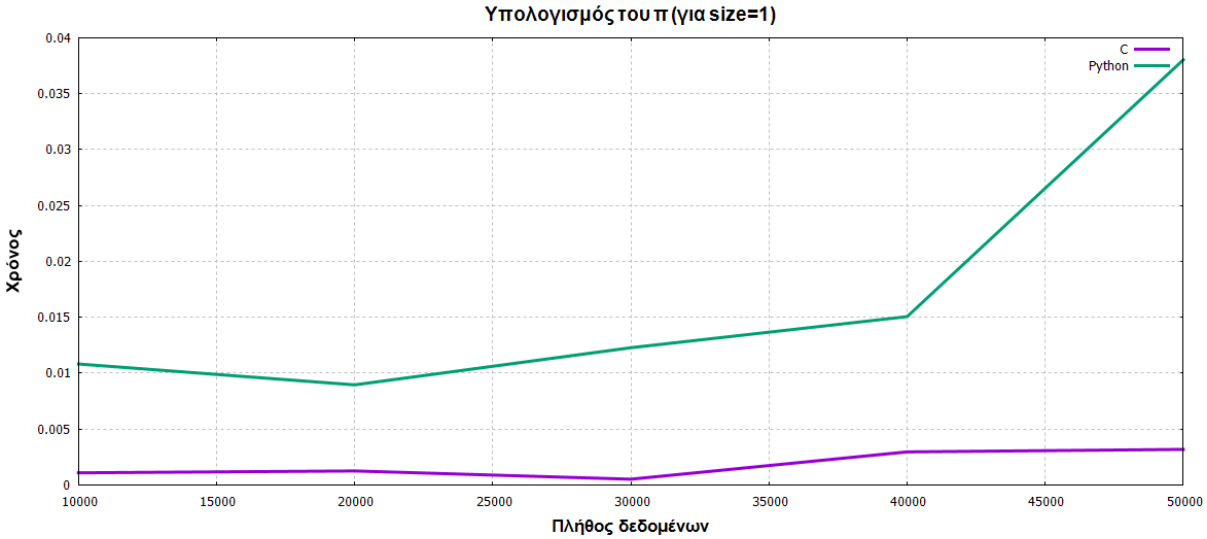
Τα παραπάνω προγράμματα εκτελέστηκαν για όγκο δεδομένων N=10000, 20000, 30000, 40000 και 50000 και χρησιμοποιήθηκαν για το σκοπό αυτό size = 1, 2, και 4 επεξεργαστές.

Προκύπτει έτσι ο παρακάτω πίνακας:

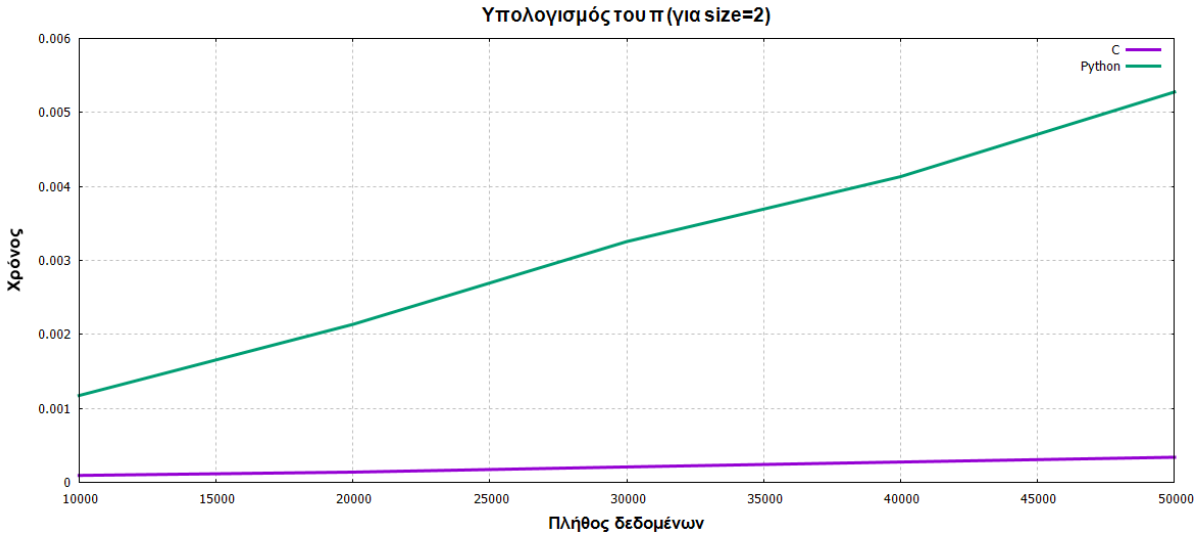
N/size	**Size=1**		**Size=2**		**Size=4**	
	C	**Python**	**C**	**Python**	**C**	**Python**
10000	0.0010933876037598	0.0108280181884766	0.0000939369201660	0.0011739730834961	0.0000622272491455	0.0008819103240967
20000	0.0012614727020264	0.0089609622955322	0.0001387596130371	0.0021359920501709	0.0000813007354736	0.0023961067199707
30000	0.0005214214324951	0.0122840404510498	0.0002086162567139	0.0032548904418945	0.0001416206359863	0.0035469532012939
40000	0.0029649734497070	0.0150630474090576	0.0002756118774414	0.0041329860687256	0.0001864433288574	0.0024690628051758
50000	0.0031898021697998	0.0380339622497559	0.0003397464752197	0.0052769184112549	0.0002334117889404	0.0029320716857910

Ακολουθούν οι γραφικές παραστάσεις για size = 1, 2 και 4.

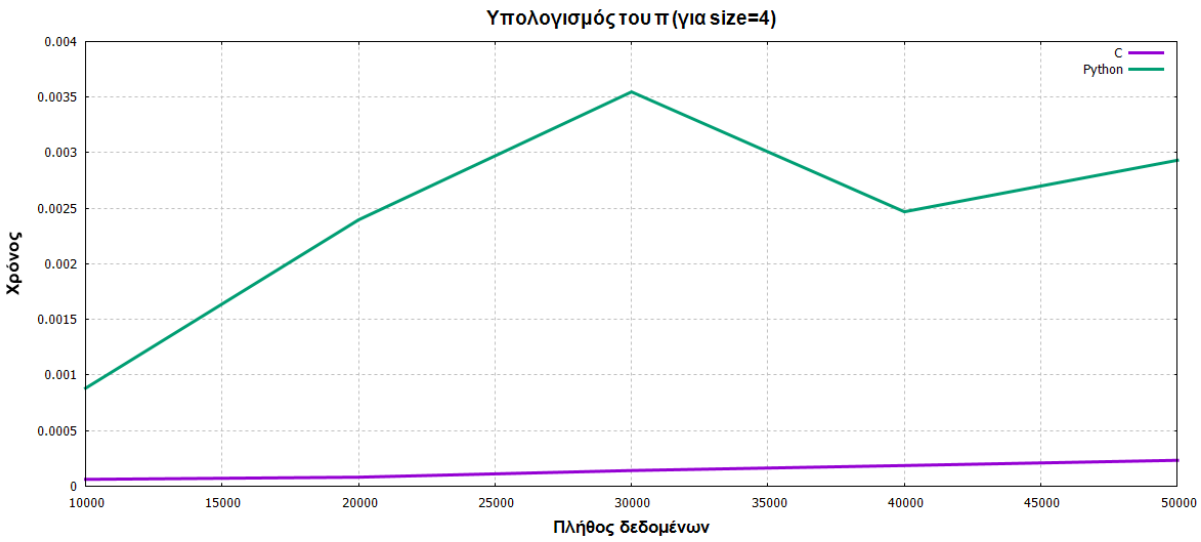
- **Για size = 1:**



- **Για size = 2:**



- **Για size = 4:**



3.3 Πρόσθεση διανυσμάτων

Λέγοντας διάνυσμα εννοούμε μονοδιάστατο πίνακα οπότε το ζητούμενο εδώ είναι η πρόσθεση δύο μονοδιάστατων πινάκων.

Ο σειριακός κώδικας εδώ έχει ως εξής:

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

Η αντίστοιχη μορφή του κατανεμημένου προγράμματος με την C είναι:

```
1  for(i=0; i<local_n;i++)
2      local_c[i]=local_a[i]+local_b[i];
```

Ενώ η αντίστοιχη μορφή με την Python είναι η ακόλουθη:

```
1  recvbuf3 = np.add(recvbuf1, recvbuf2)
```

όπου οι receive buffers εδώ αντιπροσωπεύουν τα μερικά αθροίσματα των στοιχείων που έχουν προκύψει με τη διαδικασία του scatter.

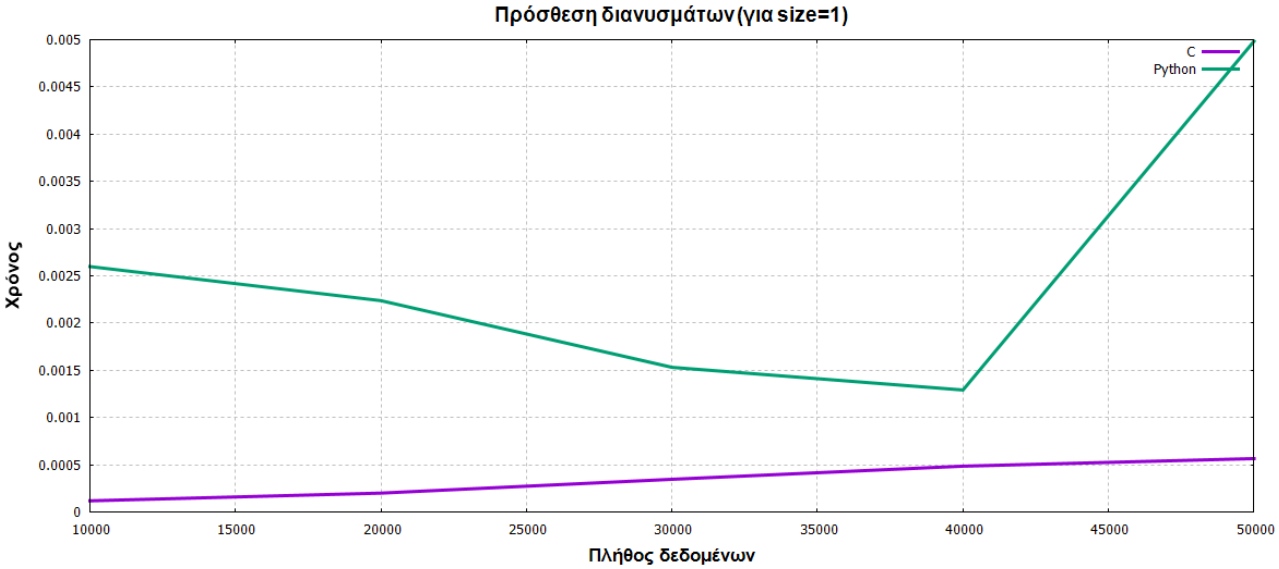
Τα παραπάνω προγράμματα εκτελέστηκαν για όγκο δεδομένων N=10000, 20000, 30000, 40000 και 50000 και χρησιμοποιήθηκαν για το σκοπό αυτό size = 1, 2, και 4 επεξεργαστές.

Προκύπτει έτσι ο παρακάτω πίνακας:

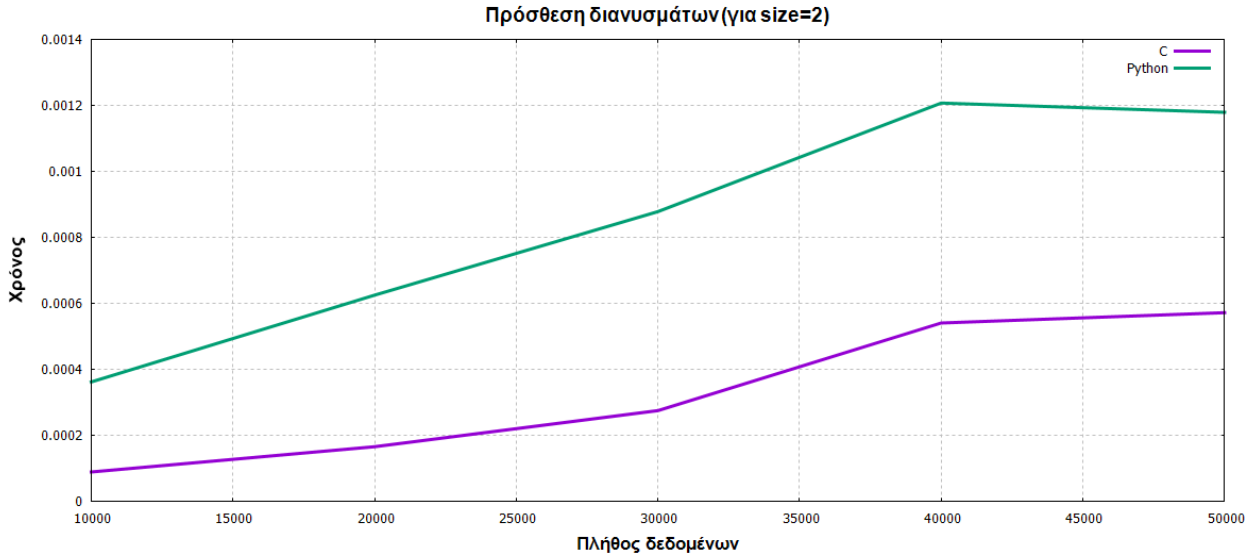
N/size	Size=1		Size=2		Size=4	
	C	Python	C	Python	C	Python
10000	0.00011801 71966553	0.00259804 72564697	0.00008893 01300049	0.00036191 94030762	0.00012946 12884521	0.00045418 73931885
20000	0.00020027 16064453	0.00223684 31091309	0.00016546 24938965	0.00062489 50958252	0.00021243 09539795	0.00088405 60913086
30000	0.00034594 53582764	0.00153112 41149902	0.00027489 66217041	0.00087809 56268311	0.00030493 73626709	0.00087189 67437744
40000	0.00048494 33898926	0.00129103 66058350	0.00054049 49188232	0.00120711 32659912	0.00049138 06915283	0.00104808 80737305
50000	0.00056505 20324707	0.00498318 67218018	0.00057172 77526855	0.00117897 98736572	0.00055670 73822021	0.00134921 07391357

Ακολουθούν οι γραφικές παραστάσεις για size = 1, 2 και 4.

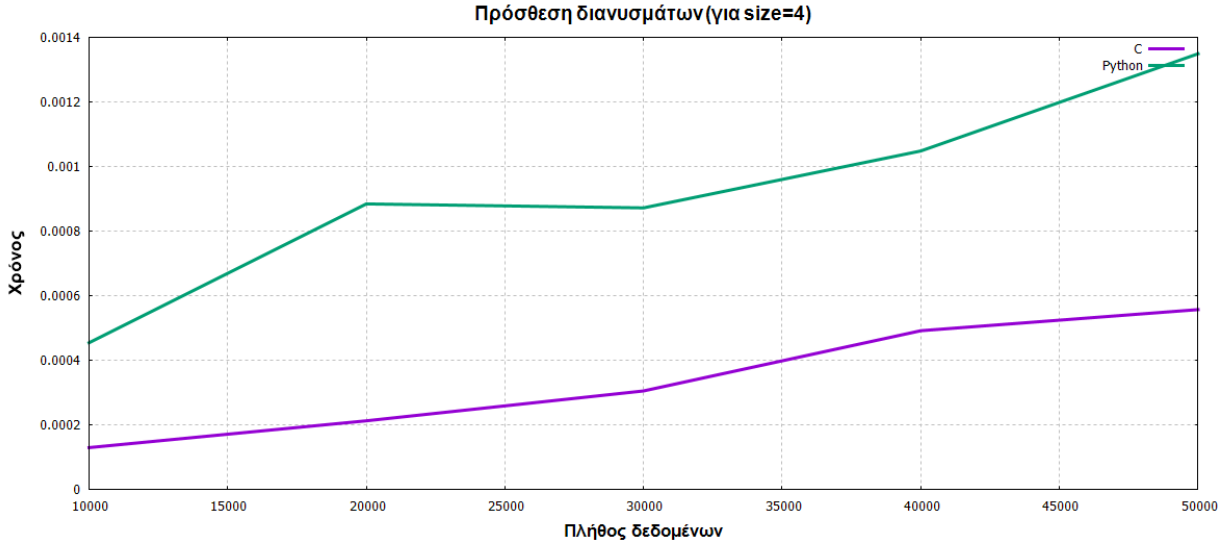
- **Για size = 1:**



- **Για size = 2:**



- **Για size = 4:**



3.4 Αναζήτηση στοιχείου σε διάνυσμα

Η σειριακή ή αλλιώς γραμμική μέθοδος αναζήτησης είναι μία απλή μέθοδος αλλά λιγότερη αποτελεσματική για πολλά στοιχεία. Στο σειριακό πρόγραμμα ψάχνουμε στο συνολικό πλήθος δεδομένων να βρούμε το προς αναζήτηση στοιχείο και μόλις βρεθεί αυξάνεται ένας μετρητής. Στα παράλληλα της C και Python αυτό γίνεται πιο γρήγορα γιατί η κάθε διεργασία αναλαμβάνει να ψάξει σε συγκεκριμένο πλήθος δεδομένων και στο τέλος τις τιμές που

προκύπτουν από κάθε διεργασία τις αθροίζουμε. Παρακάτω φαίνεται πόσο γρήγορα το κάνει η C και πόσο γρήγορα η Python.

Ο σειριακός κώδικας εδώ έχει ως εξής:

```
1  for(i=0; i<=N; i++){
2      if(Aw[i]==b)
3  counter++;
4    }
```

όπου Aw[i] είναι ο μονοδιάστατος πίνακας, b το προς αναζήτηση στοιχείο, N το συνολικό πλήθος στοιχείων του πίνακα και counter ο μετρητής.

Η αντίστοιχη μορφή του κατανεμημένου προγράμματος με την C είναι:

```
1  for(i=0; i<=N/size; i++){
2      if(Aw[i]==b)
3  counter++;
4    }
```

Ενώ η αντίστοιχη μορφή με την Python είναι η ακόλουθη:

```
1  for i in range(0,(N/size),1):
2     if(Aw[i]==b):
3        counter=counter+1
```

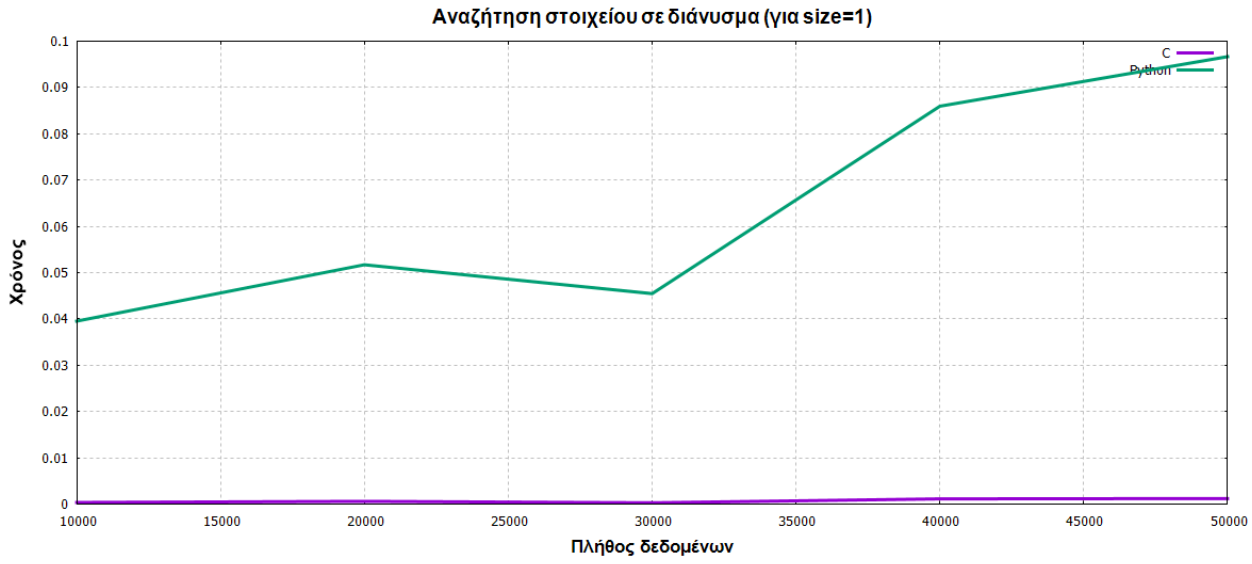
Τα παραπάνω προγράμματα εκτελέστηκαν για όγκο δεδομένων N=10000, 20000, 30000, 40000 και 50000 και χρησιμοποιήθηκαν για το σκοπό αυτό size = 1, 2, και 4 επεξεργαστές.

Προκύπτει έτσι ο παρακάτω πίνακας:

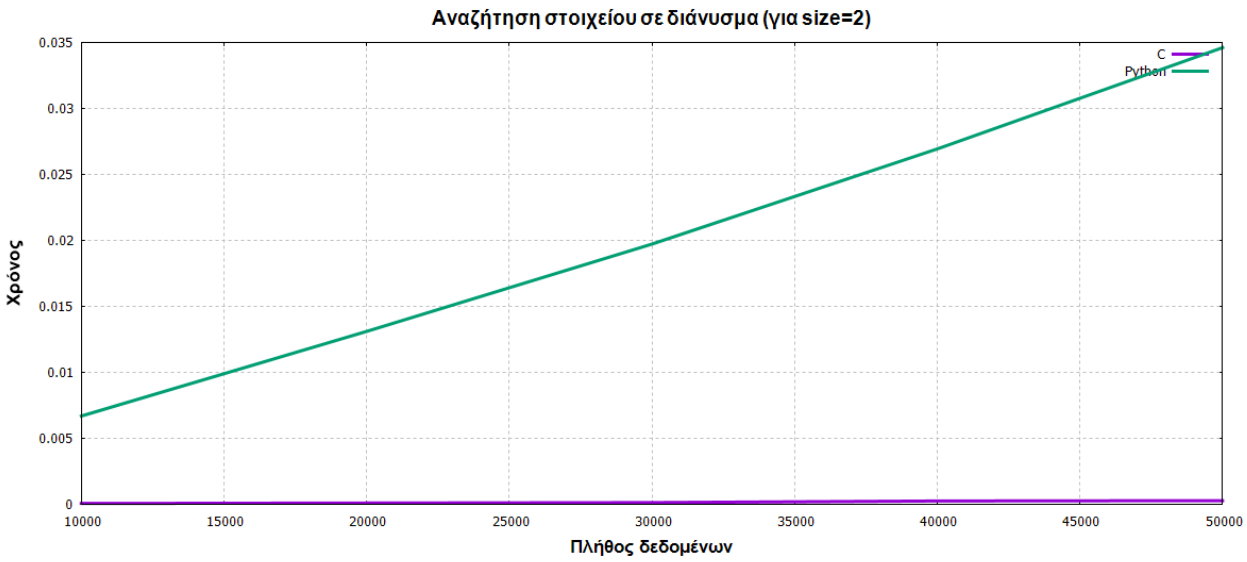
N/size	Size=1		Size=2		Size=4	
	C	Python	C	Python	C	Python
10000	0.0003583431243896	0.0395219326019287	0.0000672340393066	0.0066928863525391	0.0000813007354736	0.0039007663726807
20000	0.0005972385406494	0.0517058372497559	0.0000989437103271	0.0131058692932129	0.0001022815704346	0.0163190364837646
30000	0.0002813339233398	0.0454719066619873	0.0001366138458252	0.0197257995605469	0.0001502037048340	0.0243999958038330
40000	0.0011341571807861	0.0859501361846924	0.0002560615539551	0.0269329547882080	0.0002782344818115	0.0321760177612305
50000	0.0012042522430420	0.0966558456420898	0.0002803802490234	0.0346109867095947	0.0002741813659668	0.0752489566802979

Ακολουθούν οι γραφικές παραστάσεις για size = 1, 2 και 4.

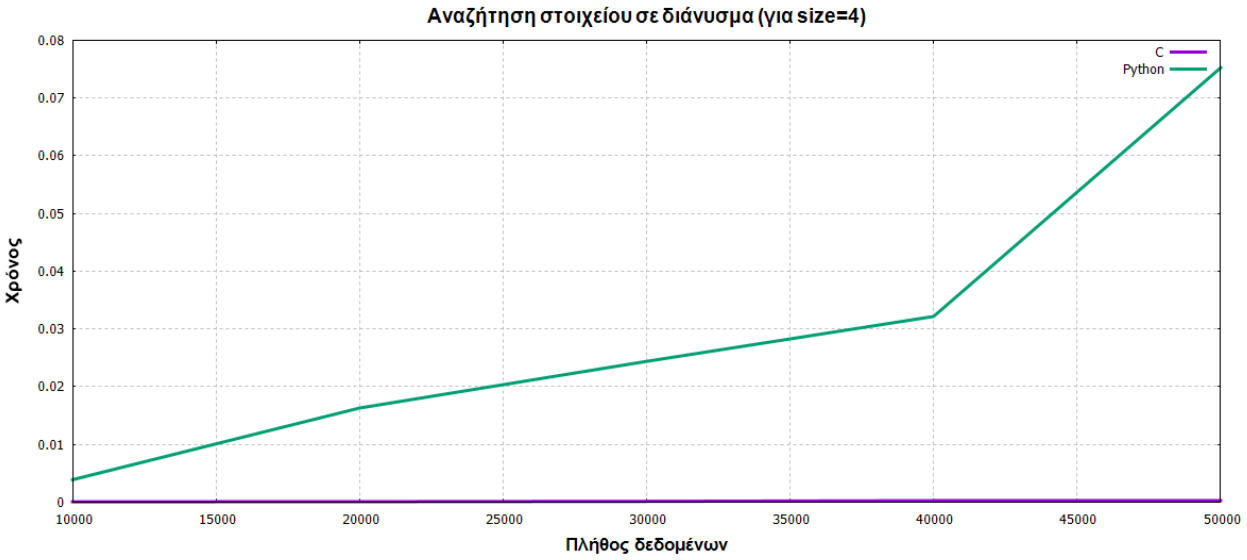
- **Για size = 1:**



- **Για size = 2:**



- **Για size = 4:**



Συμπεράσματα

C και Python! Δύο δυνατές γλώσσες προγραμματισμού με πλεονεκτήματα και μειονεκτήματα.

Μία σημαντική διαφορά τους είναι ο τρόπος που εκτελούν οι δύο αυτές γλώσσες τα προγράμματά τους. Με τα προγράμματα γραμμένα στη C, χρησιμοποιούμε έναν μεταγλωττιστή την ώρα εκτέλεσής τους. Αντίθετα με την Python χρησιμοποιούμε έναν διερμηνέα. Πιο συγκεκριμένα, ο μεταγλωττιστής είναι ένα πρόγραμμα που μετατρέπει τον πηγαίο κώδικα σε γλώσσα μηχανής που θα χρησιμοποιηθεί για την εκτέλεση του προγράμματος. Αντίθετα ένας διερμηνέας διαβάζει το πρόγραμμα που έγραψε ο χρήστης και κατευθείαν το εκτελεί. Παρόλο που αυτό συντομεύει τη διαδικασία εκτέλεσης, ο μεταγλωττιστής έχει το πλεονέκτημα ότι μπορεί να οδηγήσει σε πιο γρήγορα αποτελέσματα ως προς το χρόνο.

Μια ακόμη διαφορά είναι στην έκταση των προγραμμάτων. Τα προγράμματα που είναι γραμμένα σε C είναι πιο μεγάλα σε σχέση από αυτά με την Python. Ήταν κι ένας λόγος αυτός που ήθελα να δω την Python στην παραλληλία. Ο τρόπος σύνταξης των εντολών ήταν πολύ απλός και απίστευτα γρήγορος ο τρόπος αποτύπωσης των διαφορετικών ειδών επικοινωνίας μεταξύ των διεργασιών.

Παρατηρώντας τα διαγράμματα σε όλα τα προγράμματα, φαίνεται ξεκάθαρα ότι η Python δεν είναι κατάλληλη για υπολογισμούς υψηλής απόδοσης και οι παράλληλες δυνατότητές της είναι ακόμη κάπως ανεπαρκώς αναπτυγμένες. Μπορεί όμως να θεωρηθεί κατάλληλη για μικρούς έως και μεσαίους κώδικες και για παραγωγή εάν η επικοινωνία δεν είναι πολύ συχνή και οι επιδόσεις δεν είναι το κύριο μέλημα.

Παράρτημα

Πρώτοι αριθμοί

➢ *Πρώτοι αριθμοί (Με C)*

```
1   #include <stdio.h>
2   #include <mpi.h>
3   #include <time.h>
4
5   int main(int argc, char **argv)
6   {
7       int rank, size;
8       int prime; // λειτουργεί ως flag
9       int i, d, N;
10      int subtotal = 0, total; // μετρητής των primes
11      double ta, tt; // time
12
13      MPI_Init(&argc, &argv);
14      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15      MPI_Comm_size(MPI_COMM_WORLD, &size);
16
17      if (rank ==0 ){
18          printf("\nN=");
19          scanf("%d", &N);
20      }
21
22     ta = MPI_Wtime();  //Αρχή χρονομέτρησης
23     MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
24
```

```
25     for (i=rank+1; i<=N; i+=size) {
26         prime = 0;
27         for (d=2; d<=i/2; d++)
28             if (i % d == 0)
29                 prime = 1;
30         if (prime == 0)
31                 subtotal++;
32     }
33
34     MPI_Reduce(&subtotal, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
35
36     tt = MPI_Wtime();  //Τέλος χρονομέτρησης
37
38     if (rank == 0) {
39         printf("\nTotal number of primes between 1 and %d is: %d", N, total);
40         printf("\nExecution time: %.16f sec\n\n", tt-ta);
41     }
42
43     MPI_Finalize();
44     return 0;
45 }
```

➢ ***Πρώτοι αριθμοί (Με Python)***

```
1  from mpi4py import MPI
2  import time
3
4  comm = MPI.COMM_WORLD
5  rank = comm.Get_rank()
6  nprocs = comm.Get_size()
7
8  subtotal = 0
9  N=0
10 if (rank == 0 ):
11     N=int(input('\nN='))
12
13 ta = time.time() # Αρχή χρονομέτρησης
14
15 N=comm.bcast(N, root=0)
16
17 for i in range (rank+1,N+1,nprocs):
18     prime = 0
19     for d in range(2,(i/2)+1):
20         if (i % d) == 0:
21             prime = 1
22     if (prime == 0):
23         subtotal=subtotal+1
24
25  total=comm.reduce(subtotal,op=MPI.SUM,root=0)
26
27 tt = time.time() # Τέλος χρονομέτρησης
28
29 if (rank == 0):
30     print '\nTotal number of primes between 1 and {} is:'.format(N), total
31     print('Execution time: {:.16f} sec\n'.format(tt - ta))
```

Υπολογισμός του π = 3,14

➢ *Υπολογισμός του π (Με C)*

```
1   #include "mpi.h"
2   #include <stdio.h>
3   #include <math.h>
4   #include <time.h>
5
6   int main(int argc, char *argv[])
7   {
8       int n, rank, size;
9       int i;
10      double Pi = 3.14159265368979323848264 3;
11      double mypi, pi, h, sum, x;
12      double ta, tt;  //time
13
14      MPI_Init(&argc, &argv);
15      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16      MPI_Comm_size(MPI_COMM_WORLD, &size);
17
18     while (1) {
19       if (rank == 0 ) {
20           printf("\nEnter N (0 gia telos): ");
21           scanf("%d", &n);
22       }
23
24     ta = MPI_Wtime(); // Αρχή χρονομέτρησης
25     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
26     if ( n == 0 )
27         break;
```

```
28      else {
29          h = 1.0 / (double) n;
30          sum = 0.0;
31          for (i=rank+1; i<=n; i+=size) {
32              x = h * ((double) i - 0.5);
33             sum += (4.0 / (1 + x * x));
34          }
35          mypi = h * sum;
36           MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
37                              MPI_COMM_WORLD);
38           tt = MPI_Wtime(); // Τέλος χρονομέτρησης
39
40          if (rank == 0) {
41                printf("\nPi estimated approximately at %.16f", pi);
42                printf("\nWith error %.16f", fabs(pi - Pi));
43                printf("\nExecution time: %.16f\n", tt-ta);
44           }//if
45       }//else
46    }//while
47
48  MPI_Finalize();
49  return 0;
50  }//main
```

➢ ***Υπολογισμός του π (Με Python)***

```
1  from mpi4py import MPI
2  import time
3  import math
4
5  comm = MPI.COMM_WORLD
6  rank = comm.Get_rank()
7  nprocs = comm.Get_size()
8
9  # αριθμός βημάτων
10 nsteps=0
11 if (rank == 0 ):
12     nsteps=int(input('\nN='))
13
14 ta = time.time()  #Αρχή χρονομέτρησης
15 nsteps=comm.bcast(nsteps, root=0)
16
17 # Μέγεθος βήματος
18 dx = 1.0 / nsteps
19
20 if rank == 0:
21     # μέγεθος κάθε υποεργασίας
22     ave, res = divmod(nsteps, nprocs)
23     counts = [ave + 1 if p < res else ave for p in range(nprocs)]
24
25     # πρώτος και τελευταίος δείκτης κάθε υποεργασίας
26     starts = [sum(counts[:p]) for p in range(nprocs)]
27     ends = [sum(counts[:p+1]) for p in range(nprocs)]
28
29      # αποθήκευση των πρώτων και τελευταίων δεικτών στον data
30      data = [(starts[p], ends[p]) for p in range(nprocs)]
```

```
31 else:
32     data = None
33
34 data = comm.scatter(data, root=0)
35
36 # υπολογισμός του κάθε μερικού pi σε κάθε διεργασία
37 partial_pi = 0.0
38 for i in range(data[0], data[1]):
39     x = (i + 0.5) * dx
40     partial_pi += 4.0 / (1.0 + x * x)
41 partial_pi *= dx
42
43 partial_pi = comm.reduce(partial_pi,op=MPI.SUM,root=0)
44
45 tt = time.time()  #Τέλος χρονομέτρησης
46
47 if rank == 0:
48     print('\nPi estimated approximately at {:.16f}'.format(partial_pi))
49     print('With error {:.16f}'.format(abs(partial_pi) - math.pi))
50     print('Execution time: {:.16f} sec\n'.format(tt - ta))
```

Πρόσθεση διανυσμάτων

➢ *Πρόσθεση διανυσμάτων (Με C)*

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #include <mpi.h>
5
6   int main(int argc, char **argv)
7   {
8      int size, rank, i,local_n, N;
9      int *a = NULL;
10     int *b = NULL;
11     int *c = NULL;
12     int *local_a, *local_b,*local_c;
13     double ta, tt; // time
14
15     MPI_Init(&argc, &argv);
16     MPI_Comm_size(MPI_COMM_WORLD, &size);
17     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18
19     if (rank == 0 ) {
20        printf("\nThe size of the array: ");
21        scanf("%d", &N);
22     }
23
24     MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
25
26     local_n=N/size;
27     local_a=malloc(sizeof(int)*local_n);
```

```
28    local_b=malloc(sizeof(int)*local_n);
29    local_c=malloc(sizeof(int)*local_n);
30
31    if(rank==0) {
32          srand(time(NULL));
33          a=malloc(sizeof(int)*N);
34          b=malloc(sizeof(int)*N);
35          for(i=0;i<N;i++) {
36             a[i]=rand()%10+1;
37             b[i]=rand()%10+1;
38             printf("a[%d]=%d ",i,a[i]);
39             printf("b[%d]=%d",i,b[i]);
40             printf("\n");
41           }
42       printf("\n");
43  }
44
45  ta = MPI_Wtime(); // Αρχή χρονομέτρησης
46
47  MPI_Scatter(a,local_n,MPI_INT,local_a,local_n,MPI_INT,0,MPI_COMM_WORLD);
48  MPI_Scatter(b,local_n,MPI_INT,local_b,local_n,MPI_INT,0,MPI_COMM_WORLD);
49
50  free(a);
51  free(b);
52
53  for(i=0; i<local_n;i++)
54     local_c[i]=local_a[i]+local_b[i];
55
56  if(rank==0) {
57    printf("\nResult after gather\n");
58    c=malloc(N*sizeof(int));
59  }
60
```

```
61    MPI_Gather(local_c,local_n,MPI_INT,c,local_n,MPI_INT,0,MPI_COMM_WORLD);
62
63    tt = MPI_Wtime(); // Τέλος χρονομέτρησης
64
65    if(rank==0) {
66        for(int i=0;i<N;i++) {
67            printf("c[%d]=%d",i,c[i]);
68            printf("\n");
69        }
70        printf("\nExecution time: %.16f\n", tt-ta);
71    }
72
73    MPI_Finalize();
74    return 0;
75  }
```

➢ ***Πρόσθεση διανυσμάτων (Με Python)***

```
1  from mpi4py import MPI
2  import numpy as np
3  import time
4
5  comm = MPI.COMM_WORLD
6  rank = comm.Get_rank()
7  nprocs = comm.Get_size()
8  N=0
9
10 if (rank == 0 ):
11     N=int(input('\nN='))
12
13 N=comm.bcast(N, root=0)
14
15 #Γέμισμα πινάκων με τυχαίους αριθμούς μόνο στον rank=0
16 if rank == 0:
17     a = np.random.randint(99,size=N,dtype='i')
18     b = np.random.randint(99,size=N,dtype='i')
19     print ('First:', a)
20     print ("Second :", b)
21 else:
22     a = np.array (0, 'i')
23     b = np.array (0, 'i')
24
25 # Αρχικοποίηση των buffer arrays
26 recvbuf1 = np.empty ([N/nprocs], dtype='i')
27 recvbuf2 = np.empty ([N/nprocs], dtype='i')
28 recvbuf3 = np.empty ([N/nprocs], dtype='i')
29
30 ta = time.time()  #Αρχή χρονομέτρησης
```

```
31
32  # broadcast
33  a=comm.bcast(a,root=0)
34  b=comm.bcast(b,root=0)
35
36  #Χωρισμός πινάκων
37  comm.Scatter(a, recvbuf1, root=0)
38  comm.Scatter(b, recvbuf2, root=0)
39
40  #Άθροισμα των στοιχείων
41  recvbuf3 = np.add(recvbuf1, recvbuf2)
42
43  #Συγκέντρωση των στοιχείων στον rank=0
44  recvbuf3=comm.gather(recvbuf3, root=0)
45
46  tt = time.time() #Τέλος χρονομέτρησης
47
48  #Εμφάνιση του τελικού πίνακα μόνο από τον rank=0
49  if rank == 0:
50      print "Final array:", recvbuf3
51      print('Execution time: {:.16f} sec\n'.format(tt - ta))
```

Αναζήτηση στοιχείου σε διάνυσμα

➢ *Αναζήτηση στοιχείου σε διάνυσμα (Με C)*

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <mpi.h>
4   #include <time.h>
5
6   int main(int argc, char **argv)
7   {
8       int rank, size, counter, totalCounter, N, *A, i, b, *Aw;
9       double ta, tt; // time
10
11      MPI_Init(&argc, &argv);
12      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13      MPI_Comm_size(MPI_COMM_WORLD, &size);
14
15      if(rank==0){
16          printf("\nN= ");
17          scanf("%d", &N);
18           A = malloc(N*sizeof(int));
19
20          for(i=0;i<N;i++) {
21              A[i]=rand()%100;
22              printf("%d ", A[i]);
23          }
24
25          printf("\n\nEnter the element you are searching: ");
26          scanf("%d",&b);
27      }//if
```

```
28
29      ta = MPI_Wtime();  //Αρχή χρονομέτρησης
30
31      MPI_Bcast(&b, 1, MPI_INT, 0, MPI_COMM_WORLD);
32      MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
33
34      Aw = malloc( (N/size) * sizeof(int));
35      MPI_Scatter(A,N/size,MPI_INT,Aw,N/size,MPI_INT,0,MPI_COMM_WORLD);
36
37      counter=0;
38      totalCounter=0;
39
40      for(i=0;i<=N/size;i++){
41          if(Aw[i]==b)
42          counter++;
43      }
44
45      MPI_Reduce(&counter, &totalCounter, 1, MPI_INT, MPI_SUM, 0,
46      MPI_COMM_WORLD);
47
48      tt = MPI_Wtime();  //Τέλος χρονομέτρησης
49
50      if(rank==0) {
51          printf("The element %d found: %d fores", b, totalCounter);
52          printf("\nExecution time: %.16f sec\n\n", tt-ta);
53       }
54
55      MPI_Finalize();
56      return 0;
57  }
```

➢ ***Αναζήτηση στοιχείου σε διάνυσμα (Με Python)***

```
1  from mpi4py import MPI
2  import numpy as np
3  import time
4
5  comm = MPI.COMM_WORLD
6  rank = comm.Get_rank()
7  nprocs = comm.Get_size()
8  N=0
9  b=0
10
11 if (rank == 0 ):
12     N=int(input('\nN= '))
13
14 if rank == 0:
15     a = np.random.randint(99,size=N, dtype='i')
16     print a
17     b=int(input('\nEnter the element you are searching: '))
18 else:
19     a = np.array (0 ,'i')
20
21 N=comm.bcast(N, root=0)
22 b=comm.bcast(b, root=0)
23
24 # initialize of buffer array
25 recvbuf= np.empty ([N/nprocs], 'i')
26
27 ta = time.time()  #Arxh xronometrhshs
28
29 # broadcast
30 a=comm.bcast(a,root=0)
```

```
31
32 #splitting array "a"
33 comm.Scatter(a, recvbuf, root=0)
34
35 counter=0
36 totalCounter=0
37
38 for i in range(0,(N/nprocs),1):
39     if(recvbuf[i]==b):
40         counter=counter+1
41
42 totalCounter=comm.reduce(counter,op=MPI.SUM,root=0)
43
44 tt = time.time() #Telos xronometrhshs
45
46 if (rank == 0):
47      print 'The element {} found: {} fores'.format(b, totalCounter)
48      print('Execution time: {:.16f} sec\n'.format(tt - ta))
```

Βιβλιογραφικές αναφορές

[1] Peter Pacheco, "An Introduction to Parallel Programming", ISBN: 978-0-12-374260-5, Εκδοτικός οίκος Morgan Kauffman, 2011.

[2] Αθανάσιος Ι. Μάργαρης: «MPI – Θεωρία και Εφαρμογές», Εκδόσεις Τζιόλα, Θεσσαλονίκη 2008, ISBN 978-960-418-145-2.

[3] Τσακανίκας Β., Ταμπακάς Β., Οκτώβριος 2017. Εργαστηριακές ασκήσεις : Εισαγωγή στο Message Passing Interface – MPI. [ηλεκτρ. βιβλ.] Διαθέσιμο στο: https://openeclass.teimes.gr/modules/document/file.php/CIED393/%CE%95%CF%81%CE%B3%CE%B1%CF%83%CF%84%CE%B7%CF%81%CE%B9%CE%B1%CE%BA%CE%AE%20%CE%86%CF%83%CE%BA%CE%B7%CF%83%CE%B7%2001%20-%20%CE%95%CE%B9%CF%83%CE%B1%CE%B3%CF%89%CE%B3%CE%AE%20%CF%83%CF%84%CE%BF%20MPI.pdf.

[4] Εισαγωγή στον παράλληλο προγραμματισμό (2020, Ιανουαρίου 12). Ανακτήθηκε από https://hpc.it.auth.gr/parallel-intro/.

[5] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack J. Dongarra. MPI: The Complete Reference. Scientific and Engineering Computation. The MIT Press, second edition, 1998. Η πρώτη έκδοση διατίθεται και σε ηλεκτρονική μορφή εδώ: http://www.netlib.org/utk/papers/mpi-book/mpi-book.html.

[6] Μαργαρίτης Κ. Γ., Εργαστηριακές σημειώσεις: Προγραμματισμός Παράλληλων Υπολογιστών. Προσαρμογή από το μάθημα του Barry Wilkinson ITCS 4145/5145, 2006, Cluster Computing Univ. of North at Charlotte. Διαθέσιμο στο: http://www.it.uom.gr/teaching/wilkinson/.

[7] Neil MacDonald, Elspeth Minty, Tim Harding, and Simon Brown. Writing Message-Passing Parallel Programs with MPI: A Two Day course. Edinburgh Parallel Computing Centre, The University of Edinburgh.

[8] Δημακόπουλος, Β. Παράλληλα συστήματα και προγραμματισμός. [ηλεκτρ. βιβλ.] Αθήνα: Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών. Διαθέσιμο στο: https://repository.kallipos.gr/handle/11419/3209.

[9] Κοτρώνης Γ. Θεοχάρης Χ., Εργαστηριακές σημειώσεις: Παράλληλα Συστήματα. Διαθέσιμο στο: http://cgi.di.uoa.gr/~cotronis/ParSys/.

[10] Zhiliang Xu, Εργαστηριακές σημειώσεις: Advanced Scientific Computing. Διαθέσιμο στο: https://www3.nd.edu/~zxu2/acms60212-40212/Lec-08.pdf.

[11] Δημακόπουλος, Β. Παράλληλα συστήματα και προγραμματισμός. [ηλεκτρ. βιβλ.] Αθήνα: Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών. Διαθέσιμο στο: https://repository.kallipos.gr/handle/11419/3209.

[12] An Introduction to Parallel Programming with MPI and Python (2020, Ιανουαρίου 12). Ανακτήθηκε από https://materials.jeremybejarano.com/MPIwithPython/.

[13] L. Dalcin, R. Paz, M. Storti, and J. D'Elia, MPI for Python: performance improvements and MPI-2 extensions, Journal of Parallel and Distributed Computing, 68(5):655-662, 2008. http://dx.doi.org/10.1016/j.jpdc.2007.09.005.

[14] Lisandro Dalcin, MPI for Python, 2019. [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/mpi4py/stable/mpi4py.pdf.

[15] Distributed parallel programming in Python : MPI4PY(2020, Ιανουαρίου 12). Ανακτήθηκε από https://www.howtoforge.com/tutorial/distributed-parallel-programming-python-mpi4py/.

[16] Parallel programming in Python: mpi4py (part 1) (2020, Ιανουαρίου 12). Ανακτήθηκε από https://www.kth.se/blogs/pdc/2019/08/parallel-programming-in-python-mpi4py-part-1/.

[17] C. H. Swaroop (2015). A Byte of Python. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο, Μετάφραση: ubuntu-gr.org Team.

[18] Python MPI: Collective Operations(2020, Ιανουαρίου 12). Ανακτήθηκε από https://nyu-cds.github.io/python-mpi/05-collectives/.

[19] Python MPI: Message Passing(2020, Ιανουαρίου 12). Ανακτήθηκε από https://nyu-cds.github.io/python-mpi/02-messagepassing/.

[20] Sung Bae, Ph.D, A Hands-on Introduction to MPI Python Programming, 2014. [Online]. Available: https://www.nesi.org.nz/sites/default/files/mpi-in-python.pdf.

[21] Python Lists vs. Numpy Arrays - What is the difference? (2020, Ιανουαρίου 12). Ανακτήθηκε από https://webcourses.ucf.edu/courses/1249560/pages/python-lists-vs-numpy-arrays-what-is-the-difference?source=post_page-----f1b582a413f1----------.

[22] NumPy User Guide (2020, Ιανουαρίου 12). Ανακτήθηκε από https://numpy.org/devdocs/user/quickstart.html#the-basics.

[23] Σάββας Ηλίας (2018). Κατανεμημένος και Παράλληλος Προγραμματισμός. Σημειώσεις Τμήματος Μηχανικών Πληροφορικής ΤΕ, Λάρισα (ΤΕΙ Λάρισας).

[24] Βικιπαίδεια, ελεύθερη εγκυκλοπαίδεια(2020). Λήμμα πρώτος αριθμός. Διαθέσιμο στον δικτυακό τόπο: https://el.wikipedia.org/wiki/%CE%A0%CF%81%CF%8E%CF%84%CE%BF%CF%82%CE%B1%CF%81%CE%B9%CE%B8%CE%BC%CF%8C%CF%82. Τελευταία ενημέρωση 11 Ιανουαρίου 2020. [Πρόσβαση στις 16/1/20]

[25] Βικιπαίδεια, ελεύθερη εγκυκλοπαίδεια(2020). Λήμμα *π (μαθηματική σταθερά)*. Διαθέσιμο στον δικτυακό τόπο: https://el.wikipedia.org/wiki/%CE%A0_(%CE%BC%CE%

B1%CE%B8%CE%B7%CE%BC%CE%B1%CF%84%CE%B9%CE%BA%CE%AE_%CF%83%CF%84%CE%B1%CE%B8%CE%B5%CF%81%CE%AC). Τελευταία ενημέρωση 21 Νοεμβρίου 2019. [Πρόσβαση στις 16/1/20]