Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχ. και Μηχανικών Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

MPI: Message Passing Interface

Συστήματα Παράλληλης Επεξεργασίας

9ο Εξάμηνο

National Technical University of Athens
CSLab

Εισαγωγή

- Υπολογιστικά συστήματα υψηλής επίδοσης
  - Είναι συστοιχίες υπολογιστών (clusters)
    - Πολυπύρηνοι κόμβοι + Επιταχυντές (GPUs κ.ά.)
    - Δίκτυο διασύνδεσης υψηλής επίδοσης
    - Storage (ανά κόμβο ή σε ειδικούς κόμβους)
  - Τα clusters με τις υψηλότερες επιδόσεις στον κόσμο ονομάζονται υπερυπολογιστές (supercomputers)
    - www.top500.org
  - Χρησιμοποιούνται για την επίλυση προβλημάτων που έχουν μεγάλες απαιτήσεις:
    - Σε υπολογιστική ισχύ
    - Σε μνήμη
  - Αναπτύσσονται από τη δεκαετία του 1960
    - Cray 2 (1985): 8 διανυσματικοί επεξεργαστές, 1.9GFLOPS
    - Hitachi SR2201 (1996): 2048 επεξεργαστές, 600GFLOPS
    - Sunway TaihuLight (2016): 40960 “manycore” επεξεργαστές (256 πυρήνες/επεξεργαστή), 93 PFLOPS

Εισαγωγή

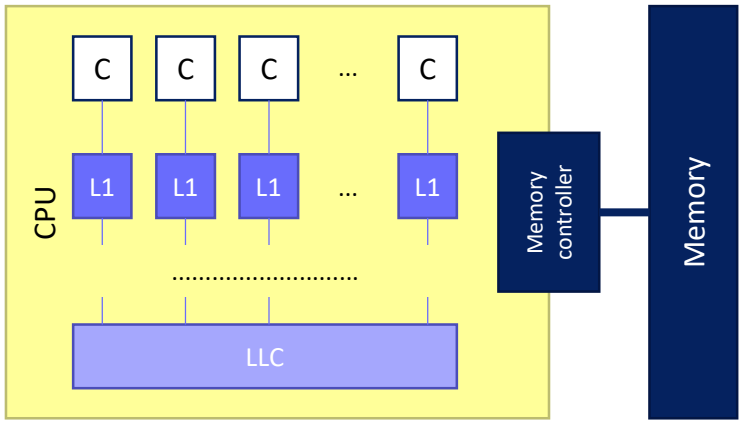
- Τα υπολογιστικά συστήματα υψηλής επίδοσης αναπτύχθηκαν και αναπτύσσονται για να καλύψουν τις ανάγκες εφαρμογών των υπολογιστικών επιστημών
  - Πρόβλεψη καιρού, Κλίμα, Αστροφυσική, Κβαντομηχανική, Δυναμική μορίων, Αεροδυναμική, Υδροδυναμική κ.ά.

- Τα υπολογιστικά συστήματα υψηλής επίδοσης επιτρέπουν την επίλυση υπολογιστικών προβλημάτων και την προσομοίωση φαινομένων
  - Σε εφικτούς χρόνους
  - Σε μεγαλύτερη κλίμακα
  - Με καλύτερη ανάλυση (resolution)
  - Με μεγαλύτερη ακρίβεια

Κύρια σημεία

- Οι συστοιχίες υπολογιστών είναι συστήματα με κατανεμημένη μνήμη και κατανεμημένο χώρο διευθύνσεων
- Ο παράλληλος προγραμματισμός σε συστήματα κατανεμημένης μνήμης διαφέρει από τον παράλληλο προγραμματισμό σε συστήματα κοινής μνήμης
- Το de facto εργαλείο προγραμματισμού είναι το MPI (Message Passing Interface)
- Η παραλληλοποίηση των επιστημονικών εφαρμογών που εκτελούνται σε συστοιχίες υπολογιστών γίνεται συνήθως με βάση τα δεδομένα τους
  - Data parallelism
  - Single Program – Multiple Data

Αρχιτεκτονική κοινής μνήμης

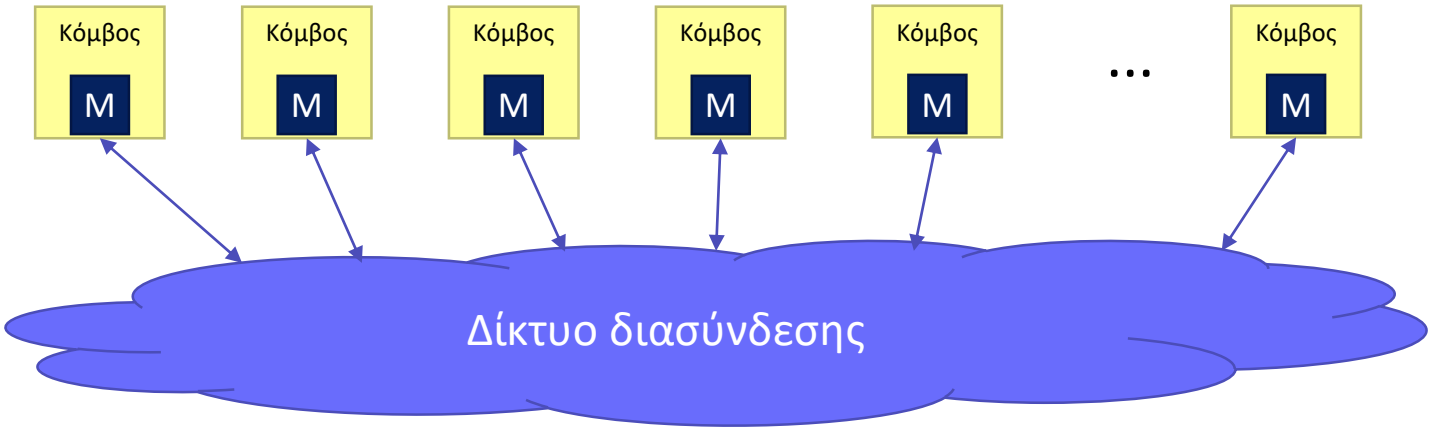


- *Παράδειγμα:* Σύγχρονοι πολυπύρηνοι επεξεργαστές
- Οι πυρήνες ενός επεξεργαστή μοιράζονται κοινή μνήμη
- Κάθε πυρήνας διαθέτει τοπική ιεραρχία κρυφών μνημών
- Διασύνδεση πυρήνων με μνήμη με διάδρομο μνήμης ή point-to-point δίκτυο διασύνδεσης

- Δύσκολα κλιμακώσιμη αρχιτεκτονική
  - Δεν κλιμακώνει το δίκτυο διασύνδεσης – γιατί;
    - Intel Broadwell – 22 cores
    - IBM Power9 SU – 24 cores

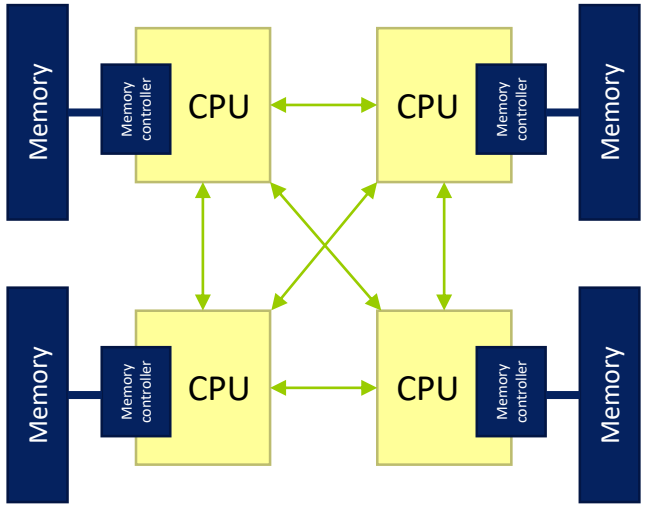
Αρχιτεκτονική κατανεμημένης μνήμης

- Κάθε «κόμβος» έχει τη δική του τοπική μνήμη και ιεραρχία τοπικών μνημών
  - Κόμβος;



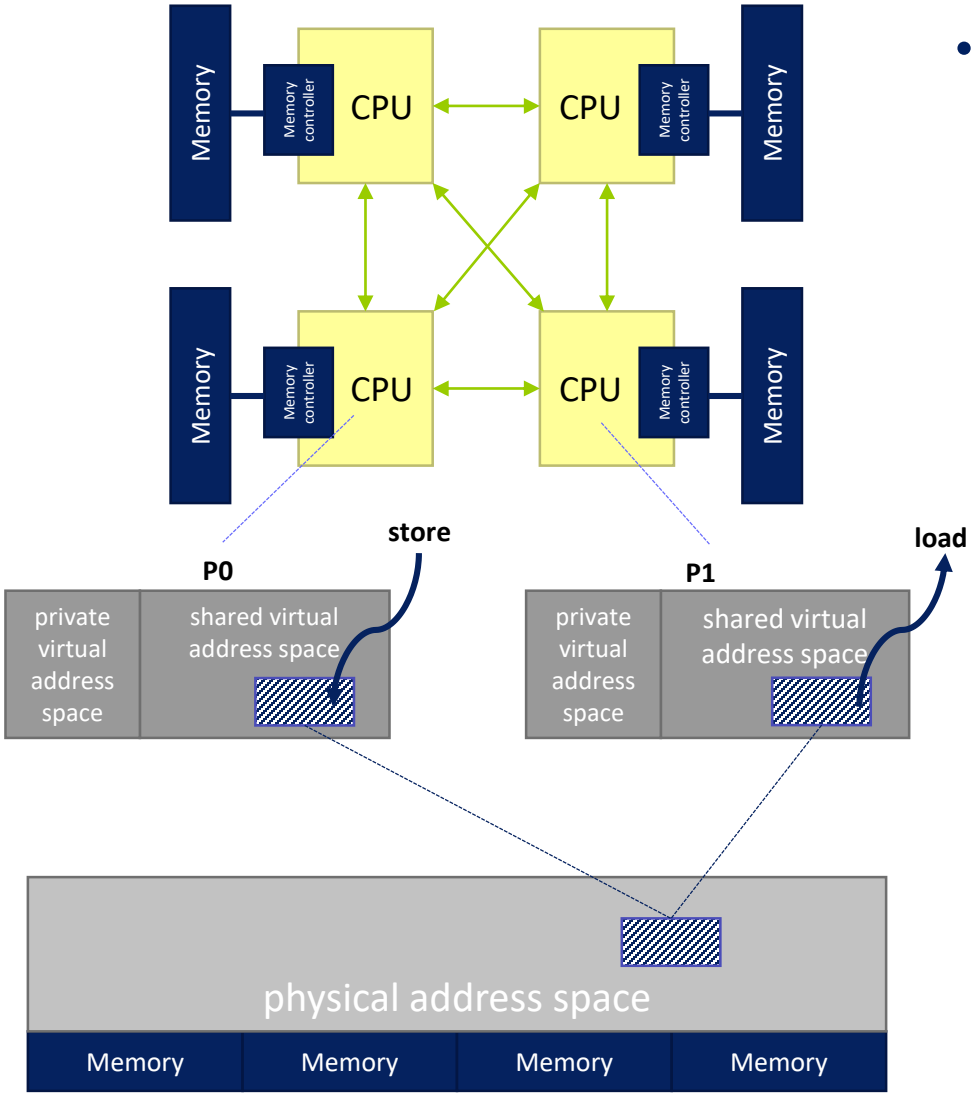
- Διασυνδέεται με τους υπόλοιπους κόμβους μέσω δικτύου διασύνδεσης
- Σε μια αρχιτεκτονική κατανεμημένης μνήμης, ο χώρος διευθύνσεων μπορεί να είναι:
  - Κατανεμημένος
  - Κοινός
  - Και τα δύο – πότε;

Αρχιτεκτονική κατανεμημένης μνήμης με κοινό χώρο διευθύνσεων



- *Παράδειγμα:*
  NUMA αρχιτεκτονική
  - Κάθε κόμβος είναι ένας επεξεργαστής με δική του μνήμη
  - Οι κόμβοι συνδέονται μέσω δικτύου διασύνδεσης
  - Οι φυσικά κατανεμημένες μνήμες χρησιμοποιούνται σαν ένας **μοναδικός κοινός** χώρος διευθύνσεων

Αρχιτεκτονική κατανεμημένης μνήμης με κοινό χώρο διευθύνσεων

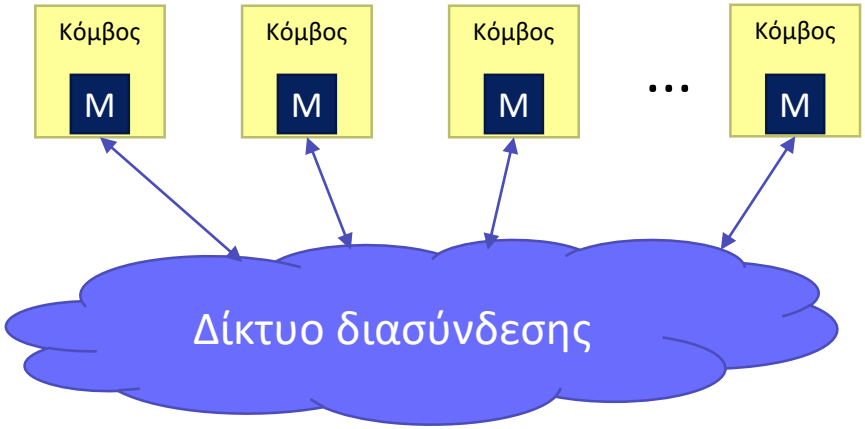


- *Παράδειγμα:*
  NUMA αρχιτεκτονική
  - Κάθε κόμβος είναι ένας επεξεργαστής με δική του μνήμη
  - Οι κόμβοι συνδέονται μέσω δικτύου διασύνδεσης
  - Οι φυσικά κατανεμημένες μνήμες χρησιμοποιούνται σαν ένας **μοναδικός κοινός** χώρος διευθύνσεων
  - Η ίδια φυσική διεύθυνση αναφέρεται στην ίδια τοποθεσία στο ίδιο κομμάτι της φυσικής μνήμης
  - Η επικοινωνία γίνεται μέσω του κοινού χώρου (loads/stores σε μοιραζόμενες μεταβλητές)
  - Υποστηρίζεται από το hardware και από το λειτουργικό σύστημα
  - Δύσκολα κλιμακώσιμη αρχιτεκτονική:
    - Λόγω δικτύου διασύνδεσης
    - Λόγω διαχείρισης μνήμης

Προγραμματισμός σε κοινό χώρο διευθύνσεων

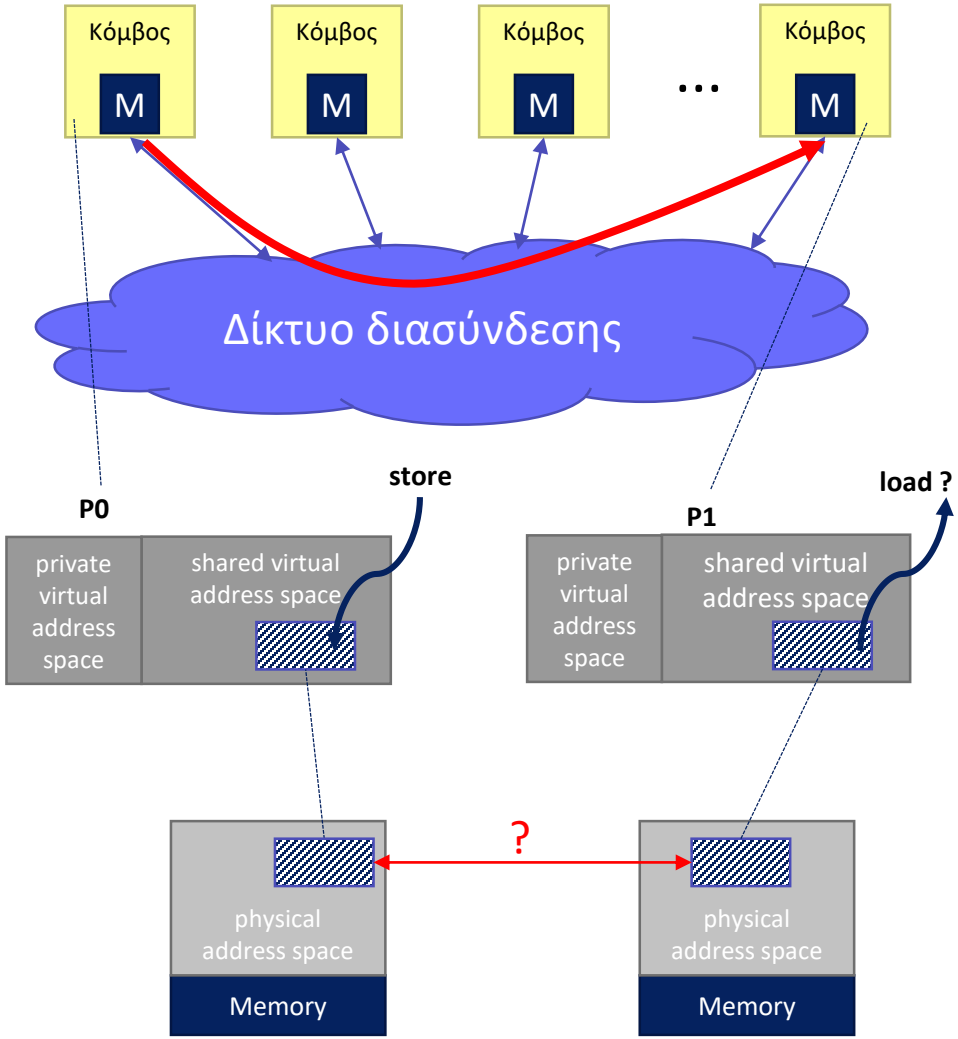
- Προγραμματιστικά μοντέλα κοινού χώρου διευθύνσεων
  - Posix Threads, OpenMP, Threading Building Blocks κ.ά.
- Οι διεργασίες έχουν πρόσβαση σε κοινό χώρο διευθύνσεων
- Χρησιμοποιούνται κανονικές εντολές ανάγνωσης / εγγραφής
  - Πιθανόν απαιτείται κάποια αρχικοποίηση του κοινού χώρου διευθύνσεων
  - Οι διευθύνσεις μπορεί να αφορούν:
    - Φυσικά κοινή μνήμη
    - Φυσικά κατανεμημένη μνήμη (απαιτείται στρώμα υλικού ή/και λογισμικού)
- Εύκολος προγραμματισμός!
  - Ο προγραμματιστής δηλώνει ιδιωτικές/κοινές μεταβλητές

Αρχιτεκτονικές κατανεμημένης μνήμης με κατανεμημένο χώρο διευθύνσεων



- *Παράδειγμα:*
  Συστοιχίες υπολογιστών
  - Κάθε κόμβος είναι ένας ή περισσότεροι επεξεργαστές με κοινή μνήμη/κοινό χώρο διευθύνσεων
  - Οι κόμβοι συνδέονται μέσω δικτύου διασύνδεσης
  - Οι φυσικά κατανεμημένες μνήμες των κόμβων αποτελούν κατανεμημένο χώρο διευθύνσεων

Αρχιτεκτονικές κατανεμημένης μνήμης με κατανεμημένο χώρο διευθύνσεων

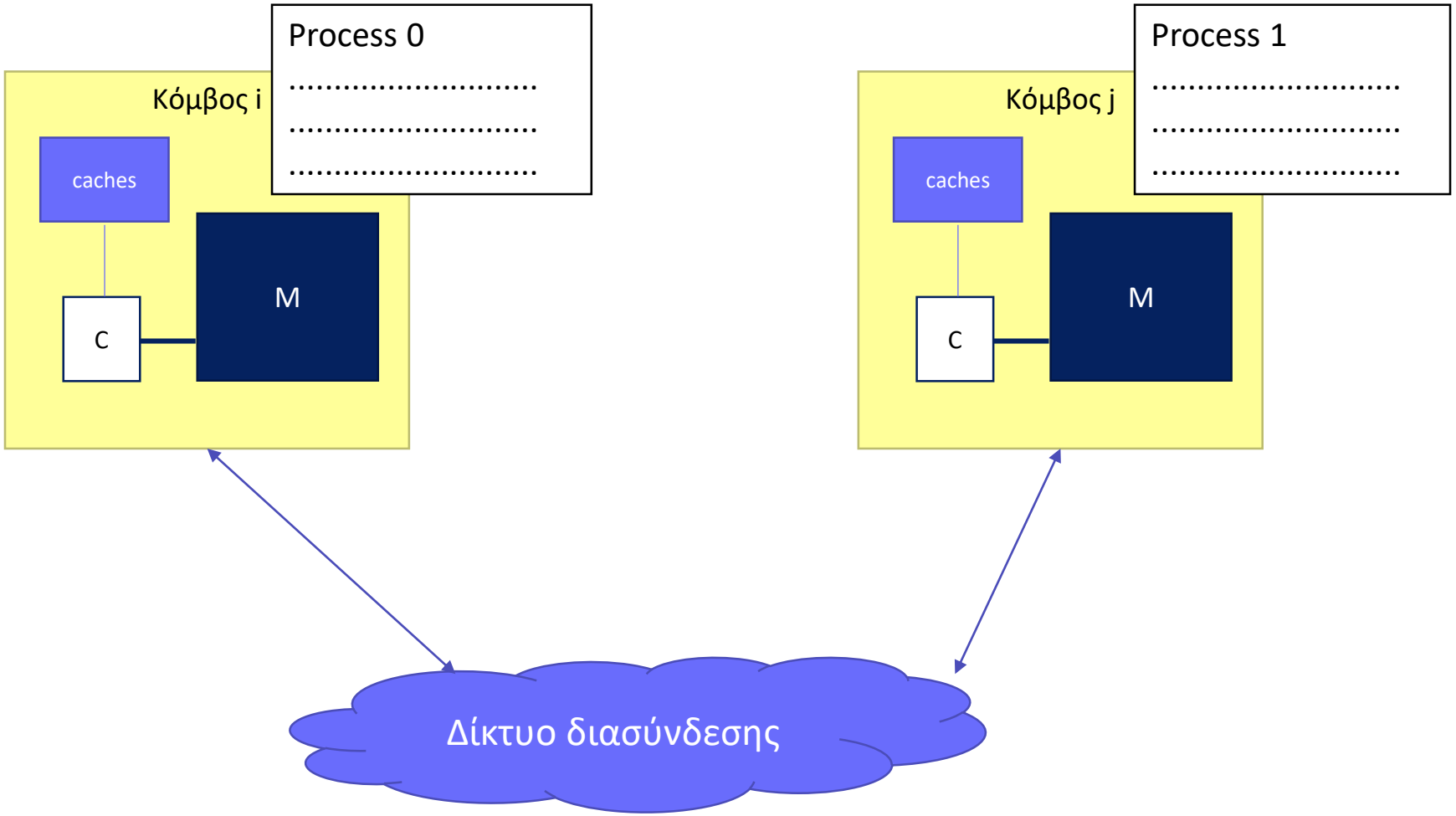


- *Παράδειγμα:*
  Συστοιχίες υπολογιστών
  - Κάθε κόμβος είναι ένας ή περισσότεροι επεξεργαστές με κοινή μνήμη/κοινό χώρο διευθύνσεων
  - Οι κόμβοι συνδέονται μέσω δικτύου διασύνδεσης
  - Οι φυσικά κατανεμημένες μνήμες των κόμβων αποτελούν κατανεμημένο χώρο διευθύνσεων
  - Δεν υπάρχουν κοινά δεδομένα
  - Η επικοινωνία γίνεται με **ανταλλαγή μηνυμάτων** (send/recv κ.ά.) μεταξύ των διεργασιών πάνω από το δίκτυο διασύνδεσης
  - Η αρχιτεκτονική κλιμακώνει σε χιλιάδες κόμβους

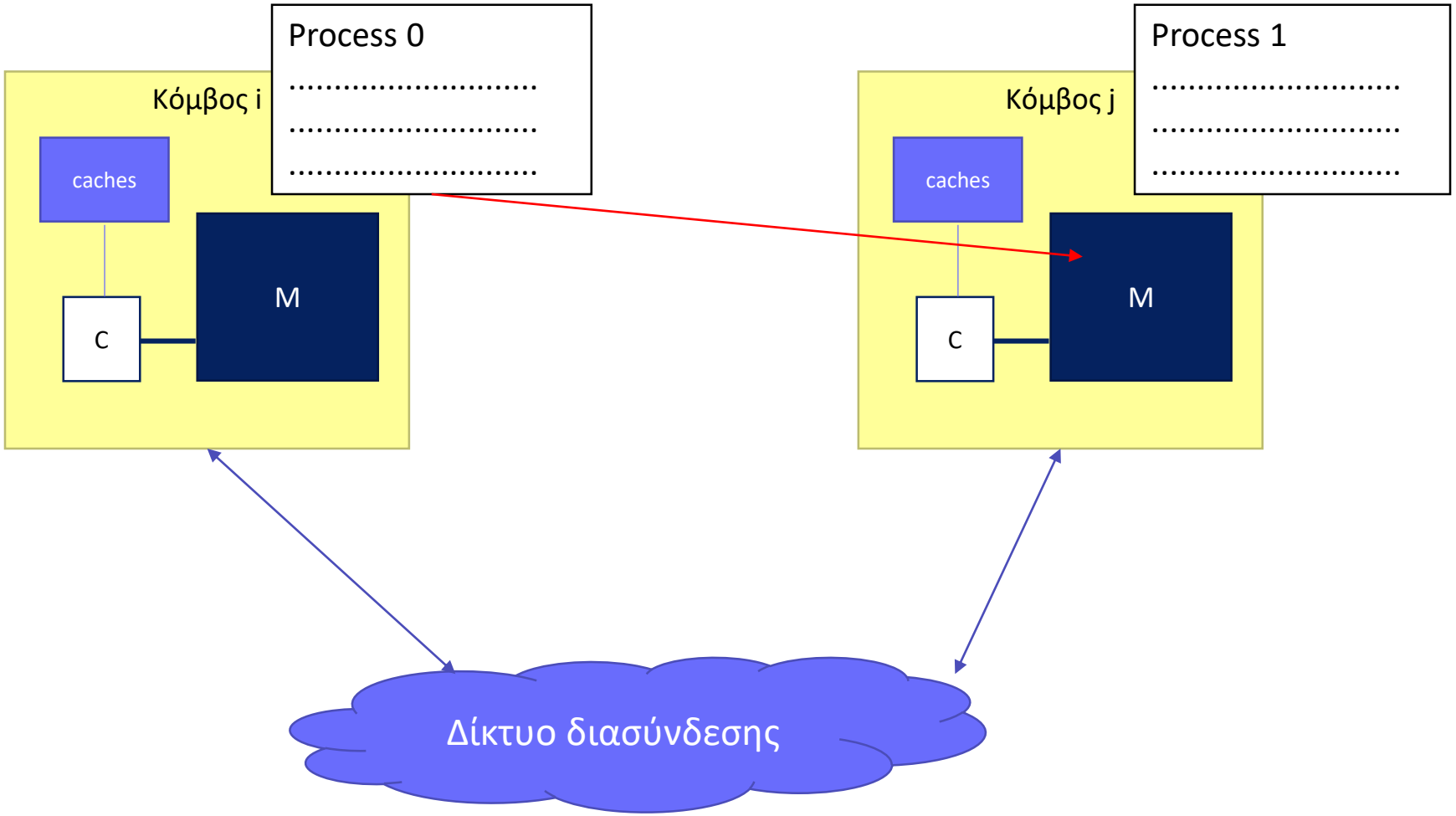
Προγραμματισμός σε κατανεμημένο χώρο διευθύνσεων

- Προγραμματιστικά μοντέλα:
  - Message Passing Interface (MPI) κ.ά.
- Η πρόσβαση σε δεδομένα που κατέχει άλλη διεργασία γίνεται με ανταλλαγή μηνυμάτων
  - Ρητές κλήσεις σε συναρτήσεις αποστολής / λήψης
- Δύσκολος προγραμματισμός!
  - Ο προγραμματιστής πρέπει να ορίσει ρητά τη μεταφορά δεδομένων μεταξύ διεργασιών

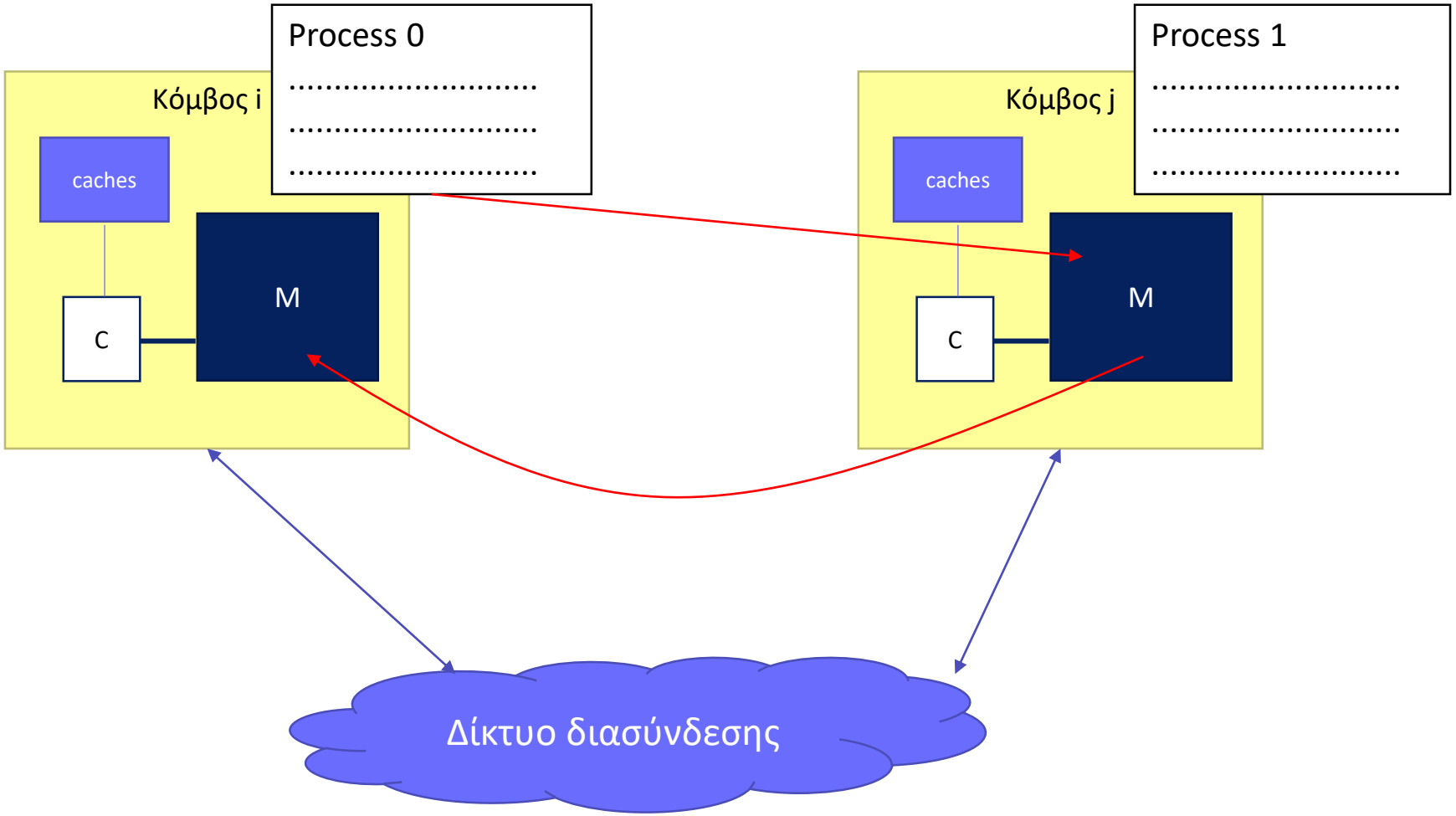
Ανταλλαγή μηνυμάτων στο MPI



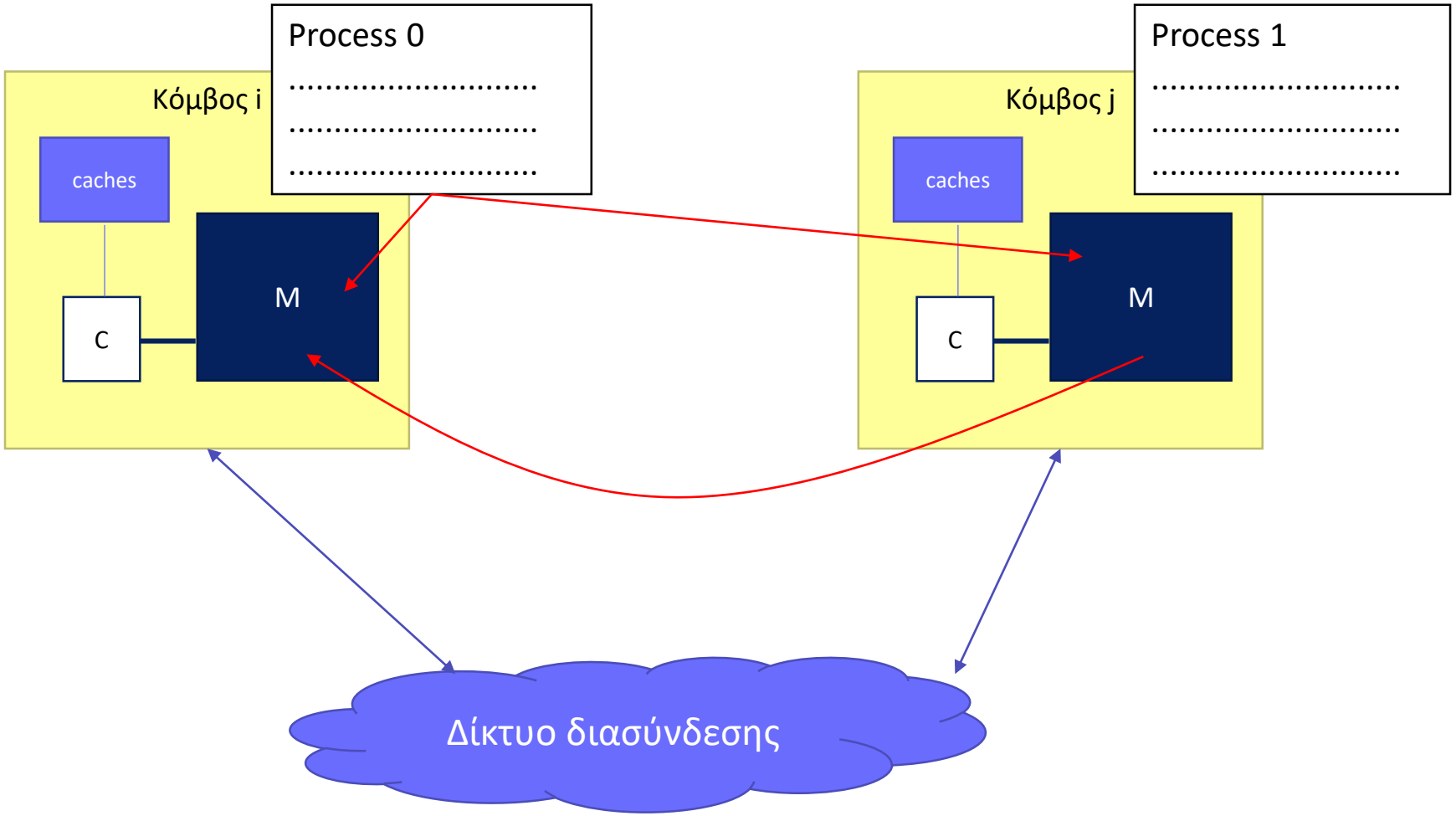
Ανταλλαγή μηνυμάτων στο MPI



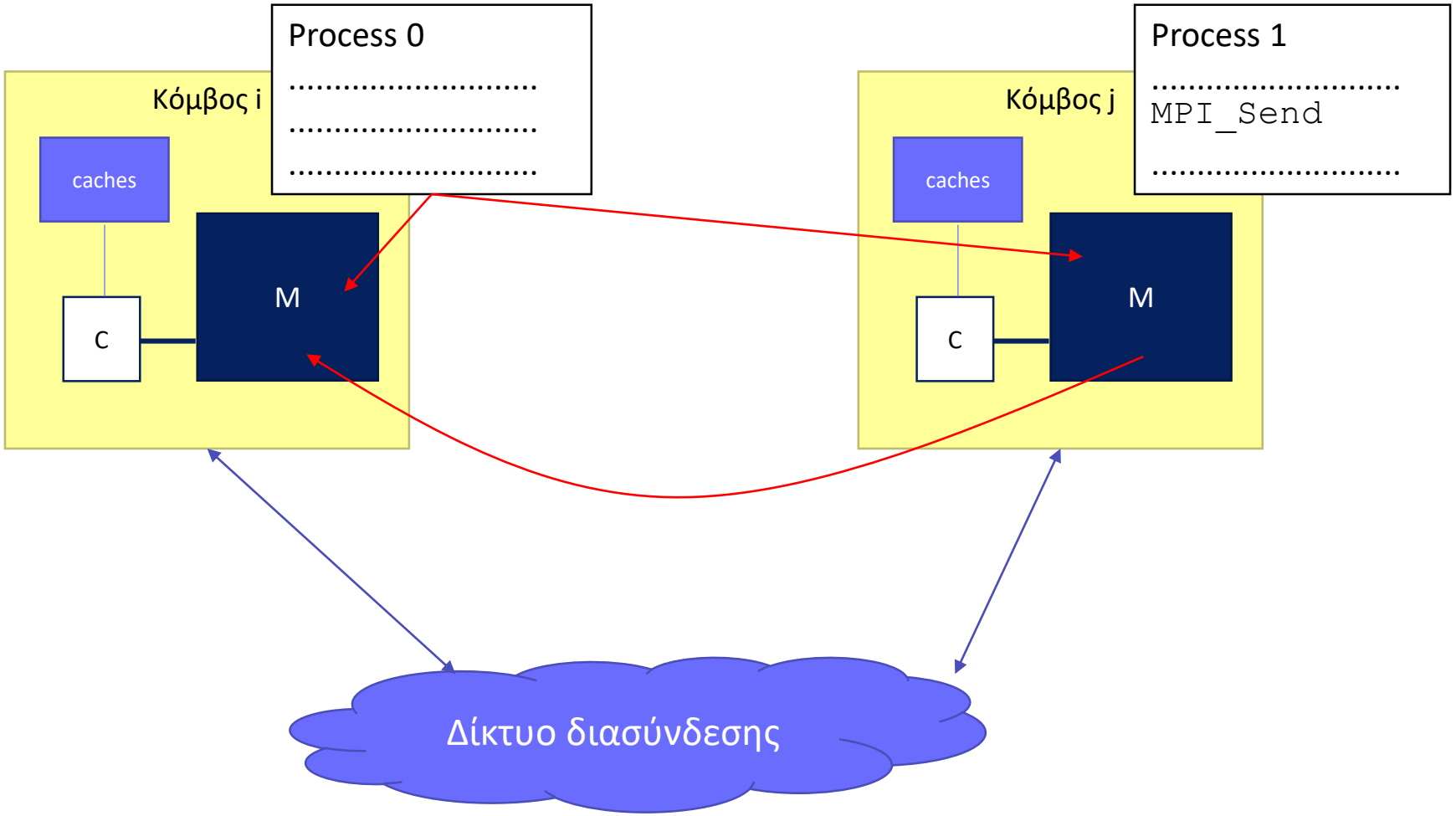
Ανταλλαγή μηνυμάτων στο MPI



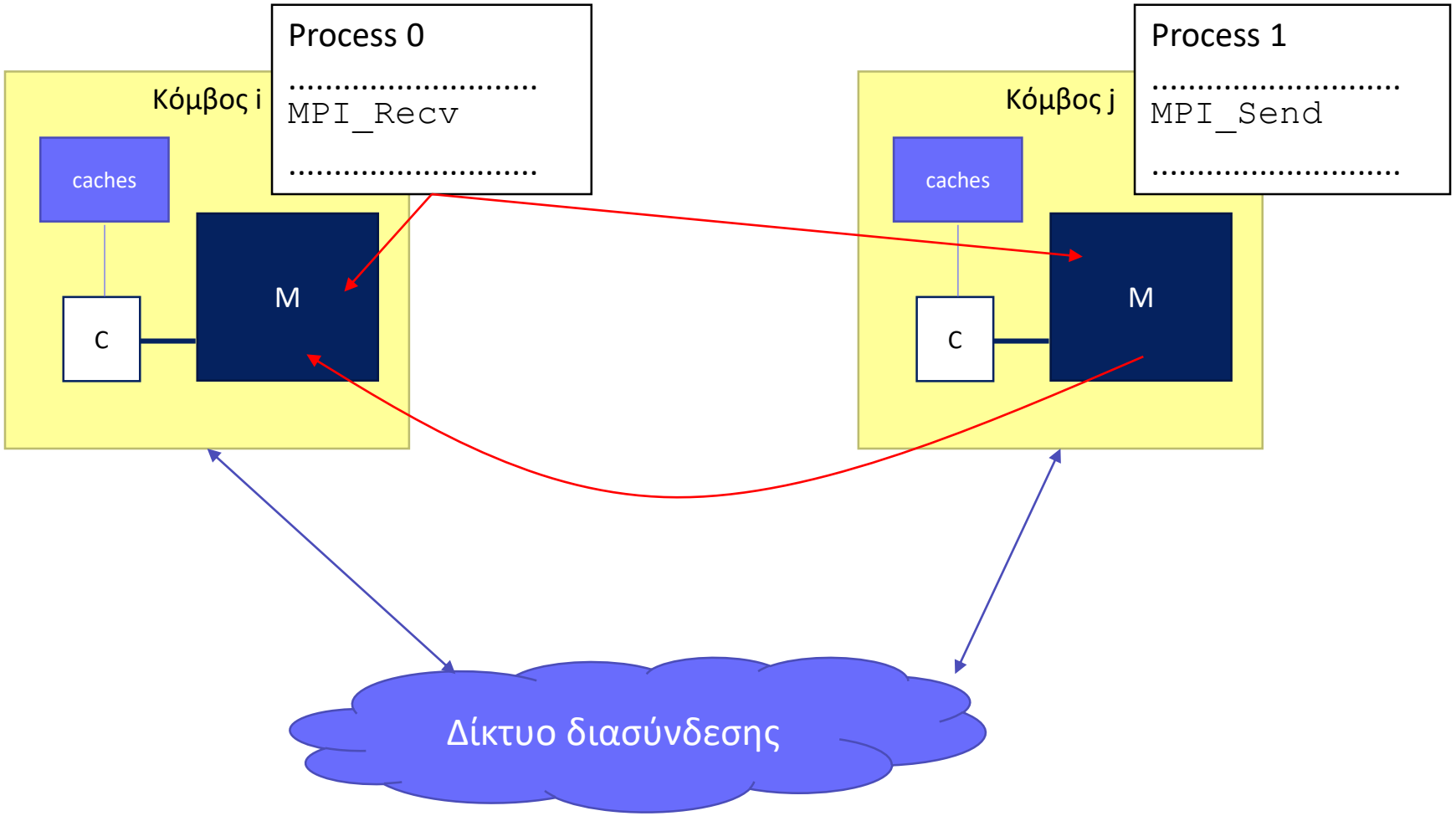
Ανταλλαγή μηνυμάτων στο MPI



Ανταλλαγή μηνυμάτων στο MPI

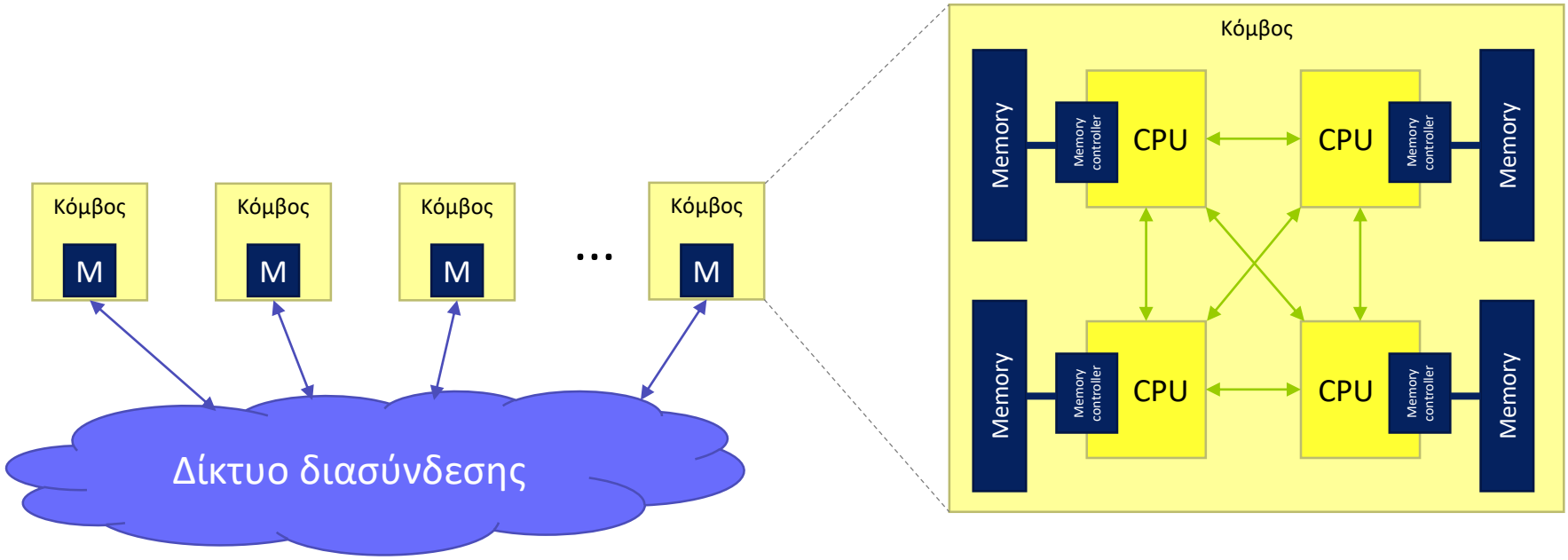


Ανταλλαγή μηνυμάτων στο MPI



Υβριδική αρχιτεκτονική

- Κοινή μνήμη εντός του κόμβου, κατανεμημένη μνήμη μεταξύ κόμβων
- Τυπική αρχιτεκτονική των σύγχρονων συστοιχιών/υπερυπολογιστών



Αρχιτεκτονικές και Προγραμματιστικά Μοντέλα από την οπτική του προγραμματιστή

		Αρχιτεκτονική	
		Κοινής μνήμης (shared memory)	Κατανεμημένης μνήμης (distributed memory)
Προγραμματιστικό μοντέλο	Κοινός χώρος διευθύνσεων (shared address space)	+ Ευκολία υλοποίησης + Προγραμματιστική ευκολία + Υψηλή επίδοση	+ Προγραμματιστική ευκολία - Δυσκολία υλοποίησης - Χαμηλή επίδοση
	Ανταλλαγή μηνυμάτων (message-passing)	+ Ευκολία υλοποίησης + Υψηλή επίδοση - Προγραμματιστική δυσκολία	+ Ευκολία υλοποίησης + Υψηλή επίδοση - Προγραμματιστική δυσκολία

Τι είναι το MPI;

- Είναι πρότυπο, όχι συγκεκριμένη υλοποίηση
- Βιβλιοθήκη ανταλλαγής μηνυμάτων
- Σχεδίαση σε στρώματα (layers)
- Σε υψηλό επίπεδο, παρέχει διεπαφή (interface) σε προγραμματιστή
- Σε χαμηλό επίπεδο, επικοινωνεί με το δίκτυο διασύνδεσης
- Υποστηρίζει C, C++, Fortran 77 και F90

Πώς προέκυψε το MPI;

- Στις αρχές του ‘90 προέκυψε η ανάγκη για πρακτικό παράλληλο προγραμματισμό σε συστήματα κατανεμημένης μνήμης (συστοιχίες υπολογιστών – clusters)
  - Εταιρείες και εργαστήρια που χρησιμοποιούσαν συστήματα κατανεμημένης μνήμης για υπολογισμούς υψηλής επίδοσης πρότειναν και διατηρούσαν τις δικές τους γλώσσες
    - Έλλειψη **μεταφερσιμότητας**!
    - Μικρές ομάδες χρηστών ανά γλώσσα/διεπαφή
  - *Εργαλεία των 90s:* Intel NX, IBM EUI, IBM CCL, PARMACS, OCCAM, PVM...

- Το MPI προέκυψε από την προσπάθεια συγκέντρωσης και τυποποίησης των καλύτερων πρακτικών από τα υπάρχοντα εργαλεία

MPI-1

Το MPI-1 προτάθηκε από το MPI forum το 1994 και όριζε:

- Point-to-point επικοινωνία
  - `MPI_Send, MPI_Recv, MPI_Rsend, MPI_Bsend, MPI_Ssend, MPI_Isend, MPI_Irecv, MPI_Wait, MPI_Waitall, MPI_Probe...`
- Blocking collectives
  - `MPI_Barrier, MPI_Bcast, MPI_Gather, MPI_Scatter, MPI_Reduce, MPI_Allreduce, MPI_Alltoall…`
- Static communicators και groups
- Τοπολογίες διεργασιών
  - Καρτεσιανή τοπολογία, Τοπολογία γράφου
- Datatypes
  - `MPI_Type_struct, MPI_Type_contiguous, MPI_Type_vector, MPI_Type_indexed`
- Διεπαφή για profiling

Υλοποιήσεις MPI

- **MPICH:** http://www.mpich.org
  - Intel MPI, Cray MPI, IBM PE MPI, TH-MPI
- MPICH-GM: http://www.myri.com/scs/
- LAM/MPI: http://www.lam-mpi.org/
- LA-MPI: http://public.lanl.gov/lampi/
- **OpenMPI:** http://www.open-mpi.org/
- SCI-MPICH: http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH/
- MPI/Pro: http://www.mpi-softtech.com/
- **MVAPICH, MVAPICH2:** http://mvapich.cse.ohio-state.edu/

Single Program Multiple Data (SPMD)

- Όλες οι διεργασίες εκτελούν το ίδιο πρόγραμμα
- Κάθε διεργασία επεξεργάζεται υποσύνολο των δεδομένων ή διαφοροποιεί τη ροή εκτέλεσης της με βάση το βαθμό (rank) που της αποδίδει το MPI
- Επιδίωξη παράλληλου προγραμματισμού:
  - Μεγιστοποίηση παραλληλίας
  - Αποδοτική αξιοποίηση πόρων συστήματος (π.χ. μνήμη)
  - Ελαχιστοποίηση όγκου δεδομένων επικοινωνίας
  - Ελαχιστοποίηση αριθμού μηνυμάτων
  - Ελαχιστοποίηση συγχρονισμού

Διεργασίες και communicators

- Ξεκινώντας ένα πρόγραμμα MPI με P διεργασίες, δημιουργείται ένας **communicator** με όνομα `MPI_COMM_WORLD` μεγέθους P διεργασιών

- Σε κάθε διεργασία αποδίδεται ένα μοναδικό **rank** (αναγνωριστικό) στο εύρος 0...P-1, όπου P το συνολικό πλήθος διεργασιών στον συγκεκριμένο communicator

- Σε γενικές γραμμές, ο communicator ορίζει ένα σύνολο από διεργασίες που μπορούν να επικοινωνούν μεταξύ τους
  - Ο `MPI_COMM_WORLD` είναι ο communicator που περιλαμβάνει όλες τις διεργασίες
  - Μπορούμε να δημιουργήσουμε οποιονδήποτε communicator με οποιεσδήποτε διεργασίες

- *Προσοχή:* Αναφερόμαστε πάντα σε διεργασίες, όχι σε επεξεργαστές

MPI Hello World

```
#include <mpi.h>

main(int argc, char **argv){
   ...
   /* Πρώτη κλήση MPI */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   printf("Hello, world from rank %d of %d\n", rank, size);

   /* Τελευταία κλήση MPI */
   MPI_Finalize();
}
```

Βασικές συναρτήσεις στο MPI

- `MPI_Init(argc,argv)`
  - Αρχικοποίηση
- `MPI_Finalize()`
  - Τερματισμός

- `MPI_Comm_rank(comm,rank)`
  - Αντιστοίχιση rank-διεργασίας σε communicator comm
- `MPI_Comm_size(comm,size)`
  - Εύρεση πλήθους διεργασιών size σε comm

- `MPI_Send(sndbuf,count,datatype,dest,tag,comm)`
  - Αποστολή μηνύματος σε διεργασία dest
- `MPI_Recv(rcvbuf,count,datatype,source,tag, comm,status)`
  - Λήψη μηνύματος από διεργασία source

MPI_Init

Αρχικοποίηση περιβάλλοντος MPI

```
int MPI_Init(int* argc, char*** argv)
```

Παράδειγμα:

```
int main(int argc, char** argv){
    …
    MPI_Init(&argc,&argv);
    …
}
```

MPI_Finalize

```
int MPI_Finalize()
```

- Τερματισμός περιβάλλοντος MPI
- Πρέπει να αποτελεί την τελευταία κλήση MPI του προγράμματος

MPI_Comm_rank

```
int MPI_Comm_rank (MPI_Comm comm, int* rank)
```

- Λήψη `rank` καλούσας διεργασίας που ανήκει στον communicator `comm`

Παράδειγμα:

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

MPI_Comm_size

```
int MPI_Comm_size (MPI_Comm comm, int* size)
```

- Λήψη πλήθους διεργασιών `size` που ανήκουν στον communicator `comm`

Παράδειγμα:

```
int size;
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Ανταλλαγή δεδομένων στο MPI

- Η ανταλλαγή δεδομένων στο MPI μοιάζει με ανταλλαγή e-mail
  - Μία διεργασία στέλνει ένα αντίγραφο των δεδομένων σε μια άλλη διεργασία (ή σε κάποιες άλλες διεργασίες), και η άλλη διεργασία το λαμβάνει
- Η επικοινωνία απαιτεί τα εξής:
  - Η διεργασία-αποστολέας πρέπει να ξέρει:
    - Πού στέλνει τα δεδομένα (το rank της διεργασίας-παραλήπτη)
    - Τι δεδομένα στέλνει (100 integers ή 200 characters, κλπ)
    - Ένα “tag” για το μήνυμα (κάτι σαν το θέμα του email, που επιτρέπει στον παραλήπτη να καταλάβει τι είναι το μήνυμα)
  - Η διεργασία-παραλήπτης πρέπει να ξέρει:
    - Ποιος στέλνει τα δεδομένα (το rank της διεργασίας-αποστολέα ή `MPI_ANY_SOURCE`)
    - Τι δεδομένα θα λάβει (μέχρι και 100 integers, κλπ)
    - Ποιο είναι το “tag” του μηνύματος (ή οποιοδήποτε tag, με `MPI_ANY_TAG`)

MPI_Send

```
int MPI_Send(
        void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm)
```

- Αποστολή μηνύματος `buf` από καλούσα διεργασία σε διεργασία με rank `dest`
- Ο πίνακας `buf` έχει `count` στοιχεία τύπου `datatype`

Παράδειγμα:

```
int message[20],dest=1,tag=55;
MPI_Send(message, 20, MPI_INT, dest, tag, MPI_COMM_WORLD);
```

MPI_Recv

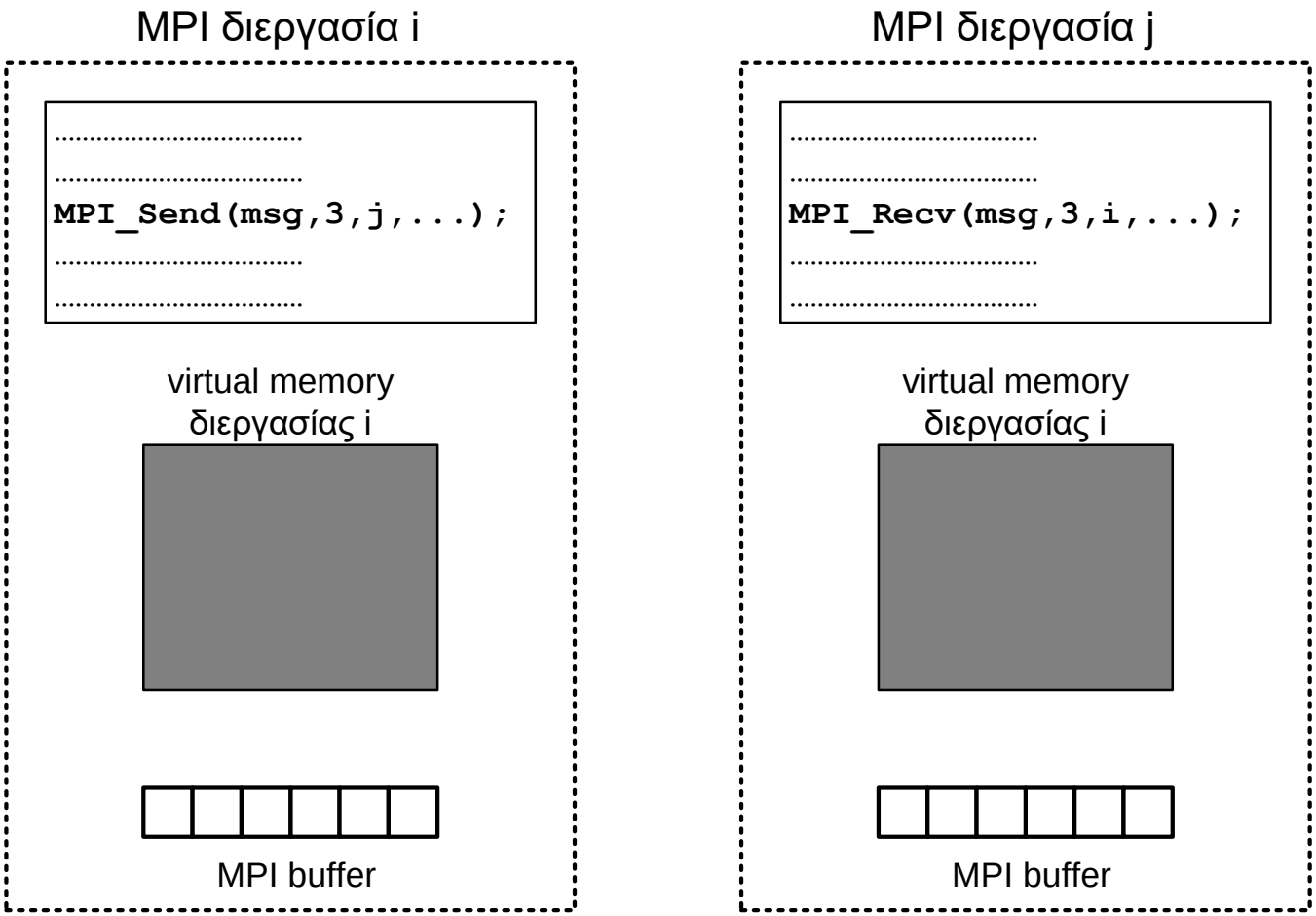
```
int MPI_Recv(
        void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm,
        MPI_Status *status)
```

- Λήψη μηνύματος από διεργασία με rank `source` και αποθήκευση σε `buf`
- Λαμβάνονται το πολύ `count` δεδομένα τύπου `datatype` (ακριβής αριθμός με `MPI_Get_count`)
- Wildcards: `MPI_ANY_SOURCE, MPI_ANY_TAG`

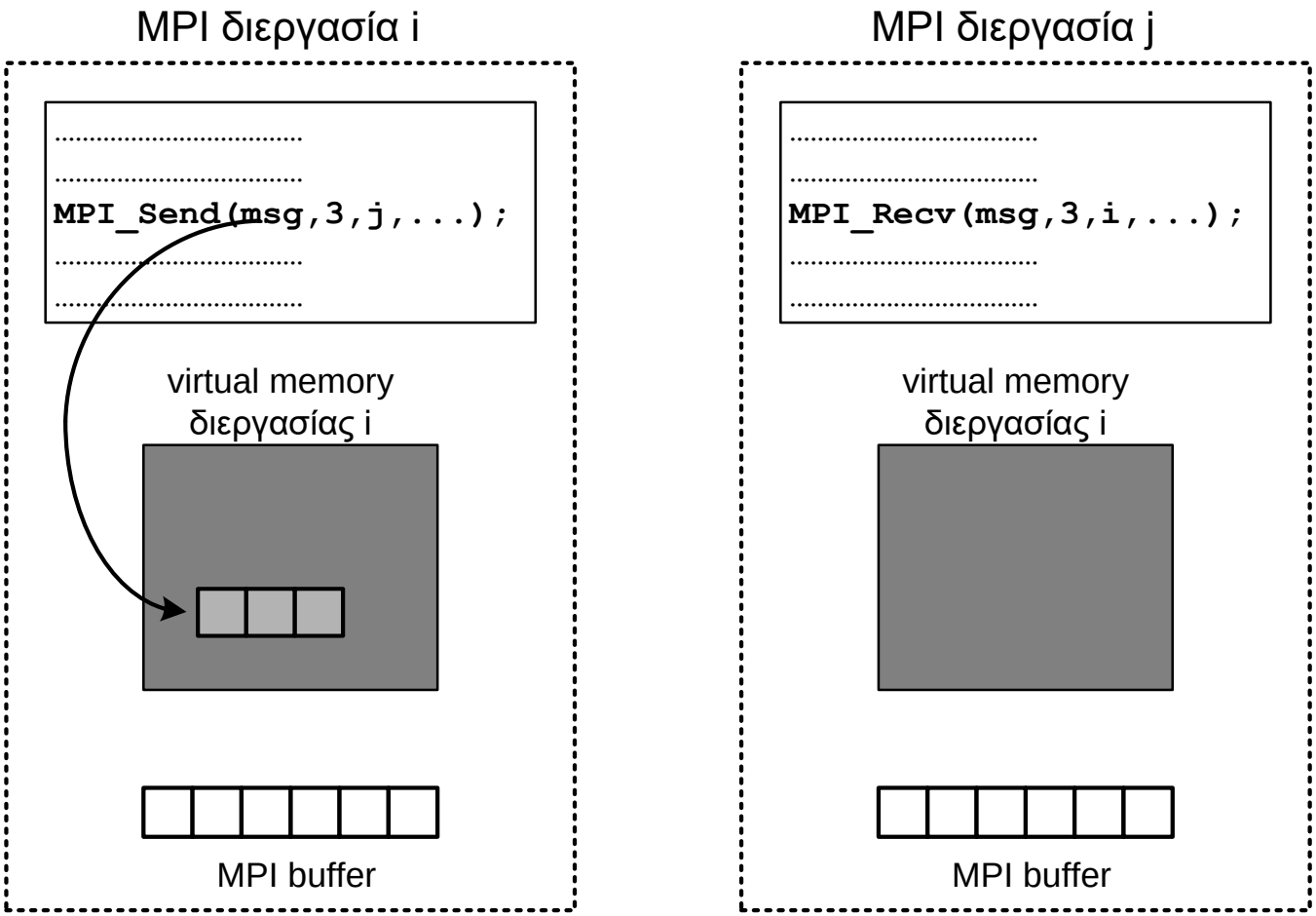
Παράδειγμα:

```
int message[50],source=0,tag=55;
MPI_Status status;
MPI_Recv(message, 50, MPI_INT, source, tag,
  MPI_COMM_WORLD,&status);
```

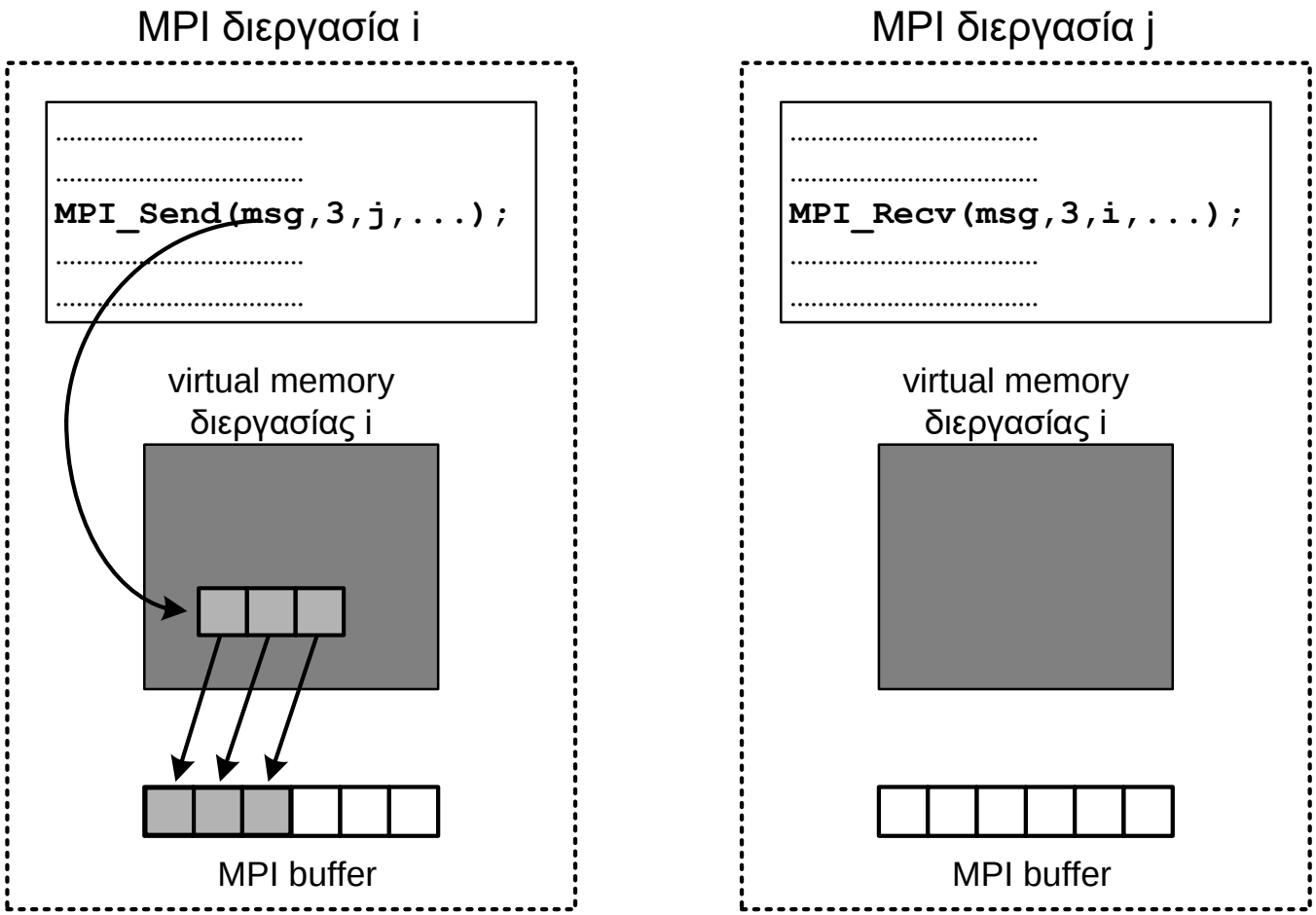
Αποστολή – λήψη στο MPI



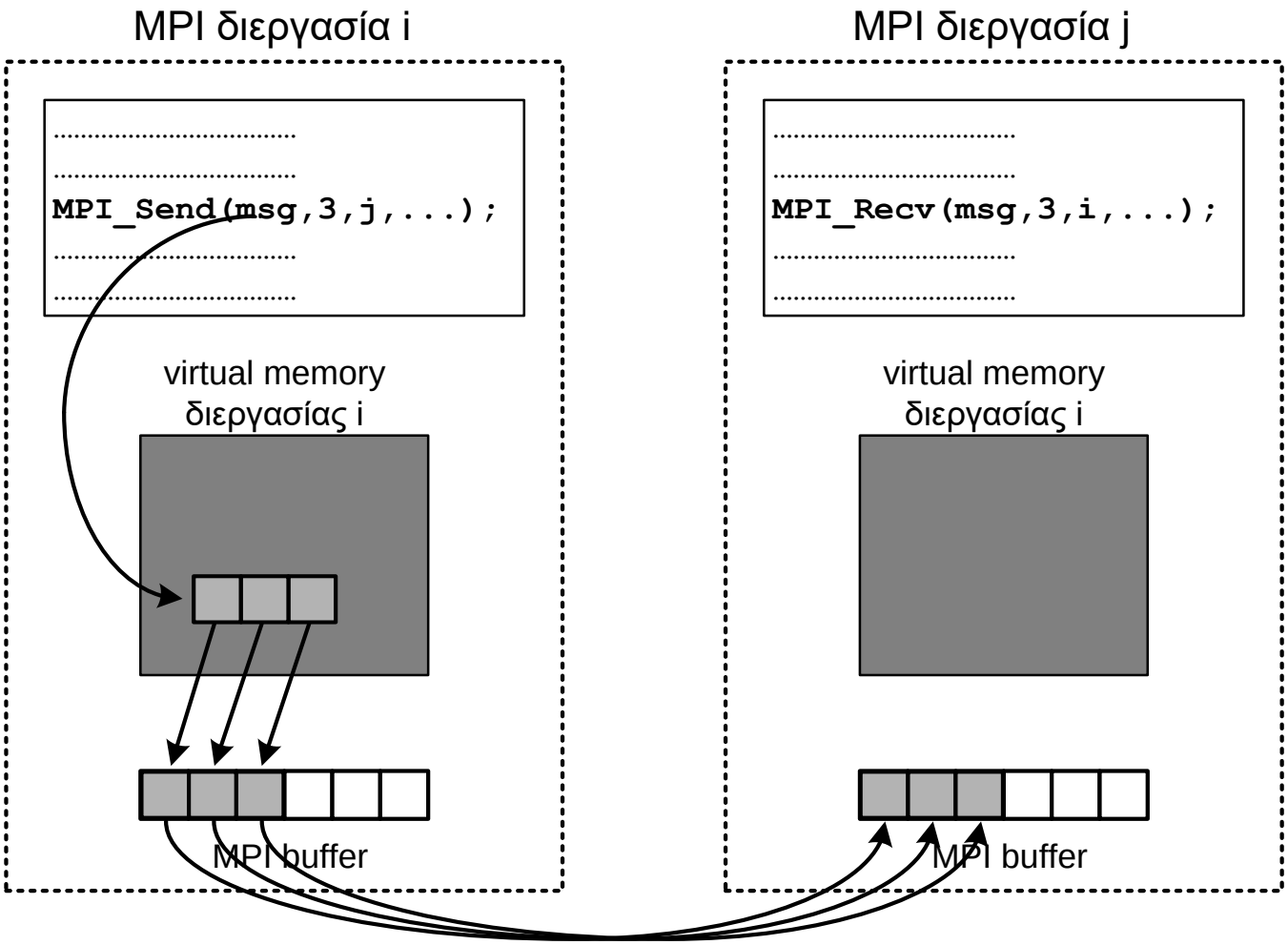
Αποστολή – λήψη στο MPI



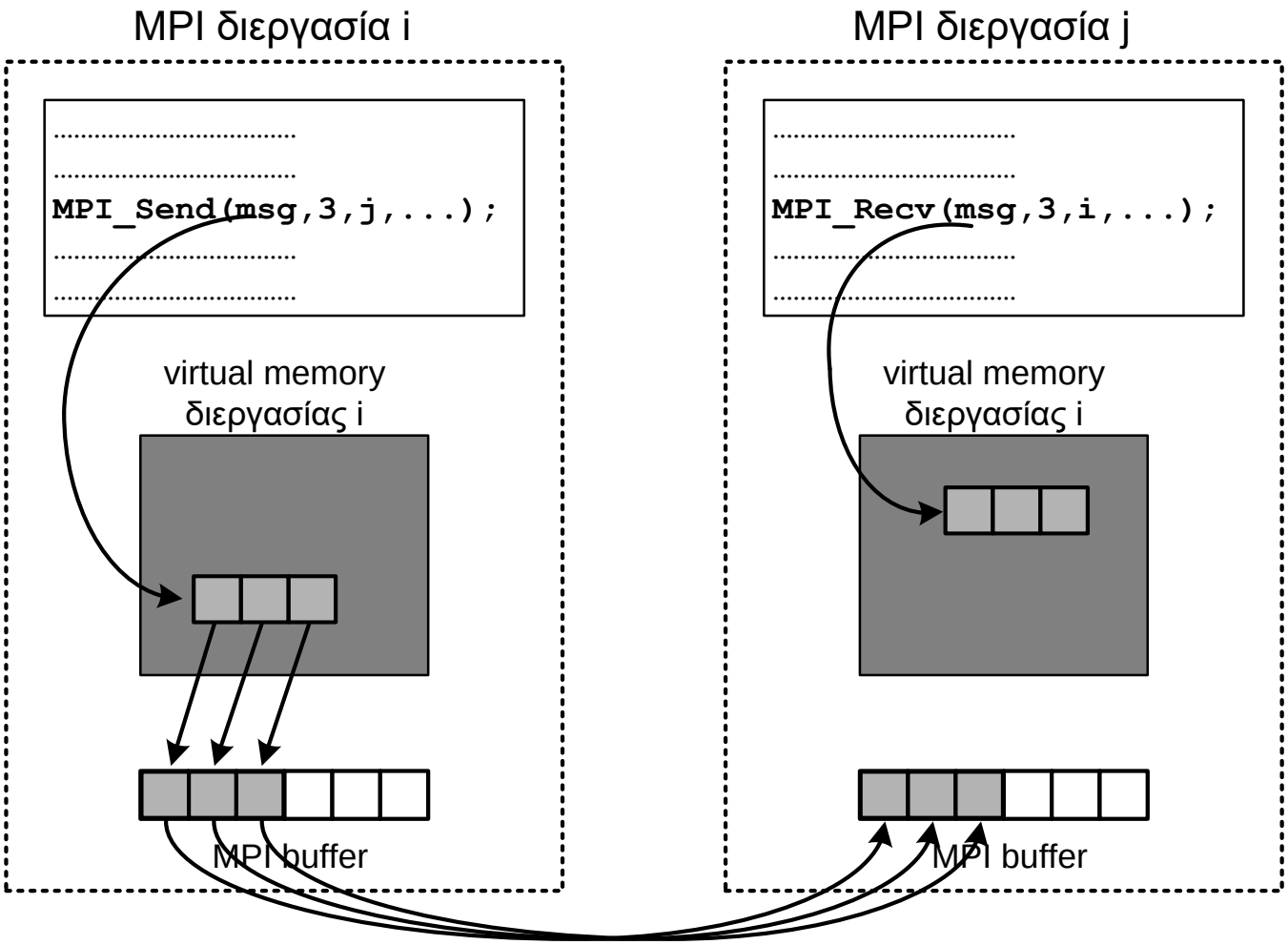
Αποστολή – λήψη στο MPI



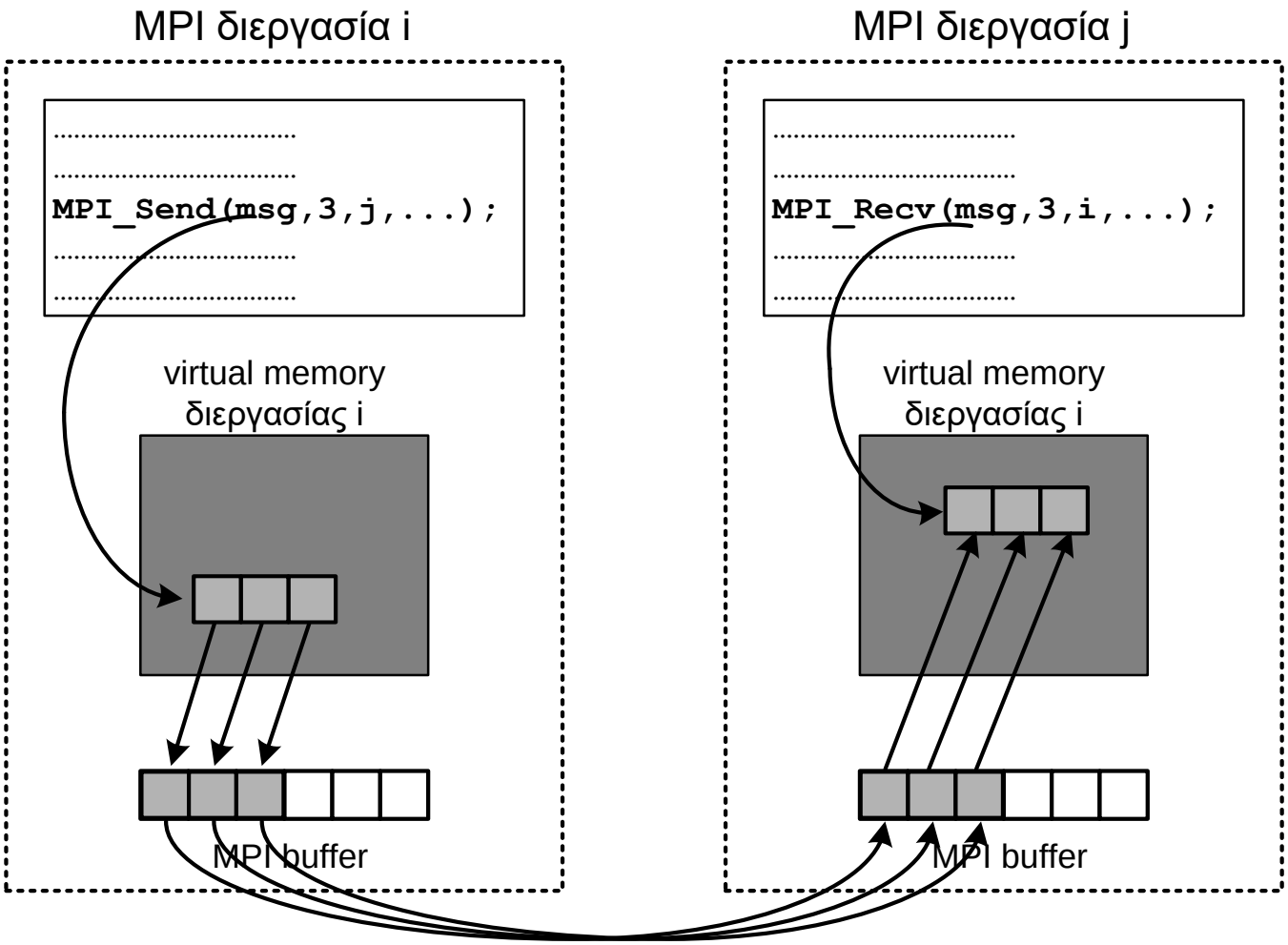
Αποστολή – λήψη στο MPI



Αποστολή – λήψη στο MPI



Αποστολή – λήψη στο MPI



MPI_Sendrecv

```
int MPI_Sendrecv(
        void *sendbuf, int sendcount, MPI_Datatype sendtype,
        int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int source, int recvtag,
        MPI_Comm comm,
        MPI_Status *status )
```

- Λήψη και αποστολή μηνύματος με μία κλήση

Παράδειγμα προγράμματος MPI

```
/* Παράλληλος υπολογισμός της παράστασης f(0)+f(1)*/
#include <mpi.h>

int main(int argc,char** argv){
    int v0,v1,sum,rank;
    MPI_Status stat;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==1)
        MPI_Send(&f(1),1,0,50,MPI_INT,MPI_COMM_WORLD);
    else if(rank==0){
        v0=f(0);
        MPI_Recv(&v1,1,1,50,MPI_INT,MPI_COMM_WORLD,&stat);
        sum=v0+v1;
    }
    MPI_Finalize();
}
```

Παράδειγμα προγράμματος MPI

```
/* Παράλληλος υπολογισμός της παράστασης f(0)+f(1)*/
#include <mpi.h>

int main(int argc,char** argv){
    int v0,v1,sum,rank;
    MPI_Status stat;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==1)
        MPI_Send(&f(1),1,0,50,MPI_INT,MPI_COMM_WORLD);
    else if(rank==0){
        v0=f(0);
        MPI_Recv(&v1,1,1,50,MPI_INT,MPI_COMM_WORLD,&stat);
        sum=v0+v1;
    }
    MPI_Finalize();
}
```

Διεργασία 1

Παράδειγμα προγράμματος MPI

```
/* Παράλληλος υπολογισμός της παράστασης f(0)+f(1)*/
#include <mpi.h>

int main(int argc,char** argv){
    int v0,v1,sum,rank;
    MPI_Status stat;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==1)
        MPI_Send(&f(1),1,0,50,MPI_INT,MPI_COMM_WORLD);
    else if(rank==0){
        v0=f(0);
        MPI_Recv(&v1,1,1,50,MPI_INT,MPI_COMM_WORLD,&stat);
        sum=v0+v1;
    }
    MPI_Finalize();
}
```

Διεργασία 1

Διεργασία 0

Επικοινωνία

- Μπορεί να είναι point-to-point ή συλλογική (collective)
- Μπορεί να είναι synchronous, buffered ή ready (ανάλογα με το τι θεωρείται ως συνθήκη επιτυχίας)
- Μπορεί να είναι blocking ή non-blocking (ανάλογα με το πότε επιστρέφει η συνάρτηση επικοινωνίας)

Συλλογική επικοινωνία

- Οι κλήσεις send/recv αντιστοιχούν σε point-to-point επικοινωνία
  - Μία διεργασία στέλνει ένα μήνυμα
  - Μία άλλη διεργασία λαμβάνει ένα μήνυμα

- Τι γίνεται όταν μία διεργασία θέλει να επικοινωνήσει με πολλές διεργασίες;
  - *Παράδειγμα:* Μία διεργασία θέλει να στείλει έναν αριθμό σε όλες τις άλλες διεργασίες (λειτουργία broadcast)

Παράδειγμα broadcast με MPI_Send/MPI_Recv

```
/* Πρόγραμμα όπου μία διεργασία λαμβάνει μία τιμή από το χρήστη και τη στέλνει σε όλες
τις άλλες διεργασίες */

int main(int argc; char **argv;)
{
    int rank, size, value, dest;
    int tag = 55;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // if root process we read the value to broadcast
    if (rank == 0) {
        printf("Enter a number to broadcast:\n"); //rank 0 reads value
        scanf("%d", &value);
    }
    else
        printf("process %d: Before, value is %d\n", rank, value);

    if (rank == 0)                                  //rank 0 sends value
        for (dest = 1 ; dest < size; dest++)
            MPI_Send(&value, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
    else                                            //all other processes receive value
        MPI_Recv(&value, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    printf("process %d: After, value is %d\n", rank, value);
    MPI_Finalize();
    return 0;
}
```

Συλλογική επικοινωνία

Παράδειγμα: Αποστολή του value στις διεργασίες 1-7 από τη 0

```
if (rank == 0)
  for(dest = 1 ; dest < size ; dest++)
    MPI_Send(&value,1,dest,tag,MPI_INT,MPI_COMM_WORLD);
  else
    MPI_Recv(&value,1,MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
```

Παράδειγμα: Αποστολή του msg στις διεργασίες 1-7 από τη 0

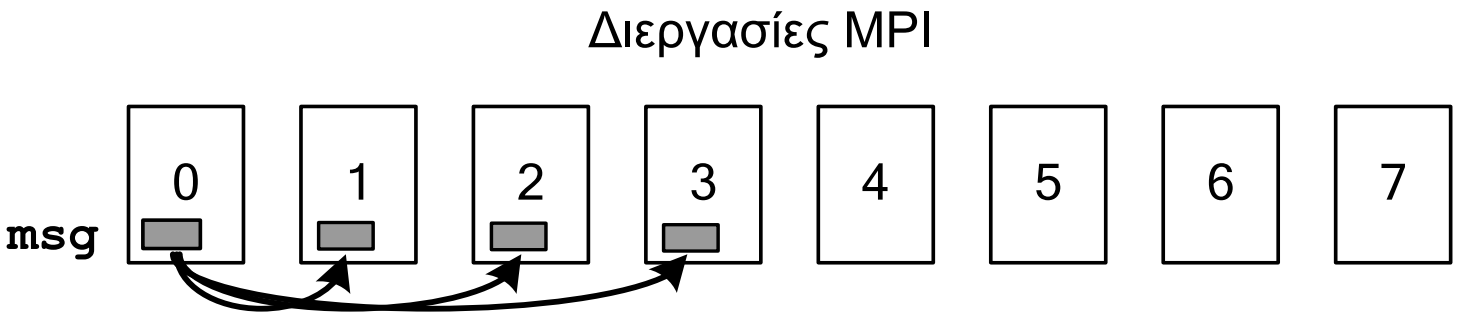
```
if (rank == 0)
  for(dest = 1 ; dest < size ; dest++)
    MPI_Send(&value,1,dest,tag,MPI_INT,MPI_COMM_WORLD);
  else
    MPI_Recv(&value,1,MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
```

Παράδειγμα: Αποστολή του msg στις διεργασίες 1-7 από τη 0

```
if (rank == 0)
  for(dest = 1 ; dest < size ; dest++)
    MPI_Send(&value,1,dest,tag,MPI_INT,MPI_COMM_WORLD);
  else
    MPI_Recv(&value,1,MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
```

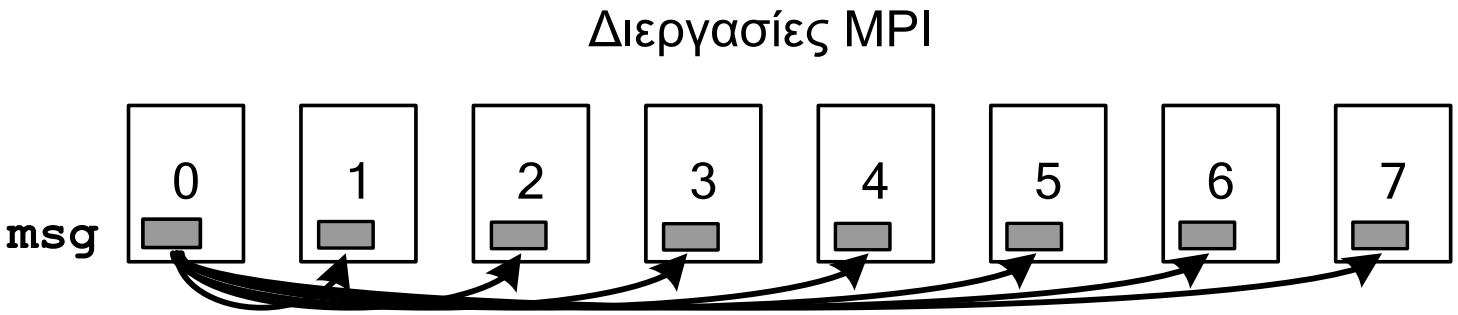
Παράδειγμα: Αποστολή του msg στις διεργασίες 1-7 από τη 0

```
if (rank == 0)
  for(dest = 1 ; dest < size ; dest++)
    MPI_Send(&value,1,dest,tag,MPI_INT,MPI_COMM_WORLD);
  else
    MPI_Recv(&value,1,MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
```



Παράδειγμα: Αποστολή του msg στις διεργασίες 1-7 από τη 0

```
if (rank == 0)
  for(dest = 1 ; dest < size ; dest++)
    MPI_Send(&value,1,dest,tag,MPI_INT,MPI_COMM_WORLD);
  else
    MPI_Recv(&value,1,MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
```



Γενικά: Για p διεργασίες έχουμε p-1 βήματα επικοινωνίας

Συλλογική επικοινωνία

- Το MPI παρέχει έτοιμες συναρτήσεις για συλλογική επικοινωνία
  - `MPI_Bcast, MPI_Reduce, MPI_Scatter` κ.ά.
- Αυτές οι συναρτήσεις καλούνται από όλες τις διεργασίες του communicator
- Παρέχουν προγραμματιστική ευκολία και καλύτερη επίδοση

Παράδειγμα broadcast με MPI_Bcast

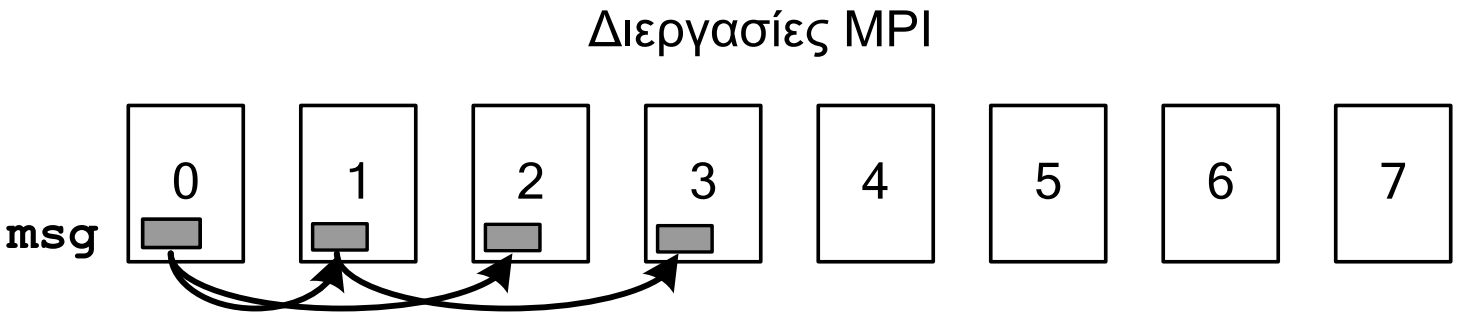
```
/* Πρόγραμμα όπου μία διεργασία λαμβάνει μία τιμή από το χρήστη και τη στέλνει σε όλες
τις άλλες διεργασίες */

int main(int argc; char **argv;)
{
    int rank, size, value, dest;
    int tag = 55;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // if root process we read the value to broadcast
    if (rank == 0) {
        printf("Enter a number to broadcast:\n"); //rank 0 reads value
        scanf("%d", &value);
    }
    else
        printf("process %d: Before, value is %d\n", rank, value);

    MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("process %d: After, value is %d\n", rank, value);
    MPI_Finalize();
    return 0;
}
```

Παράδειγμα: Αποστολή του value στις διεργασίες 1-7 από τη 0

```
MPI_Bcast(&value,1,MPI_INT,0,MPI_COMM_WORLD);
```

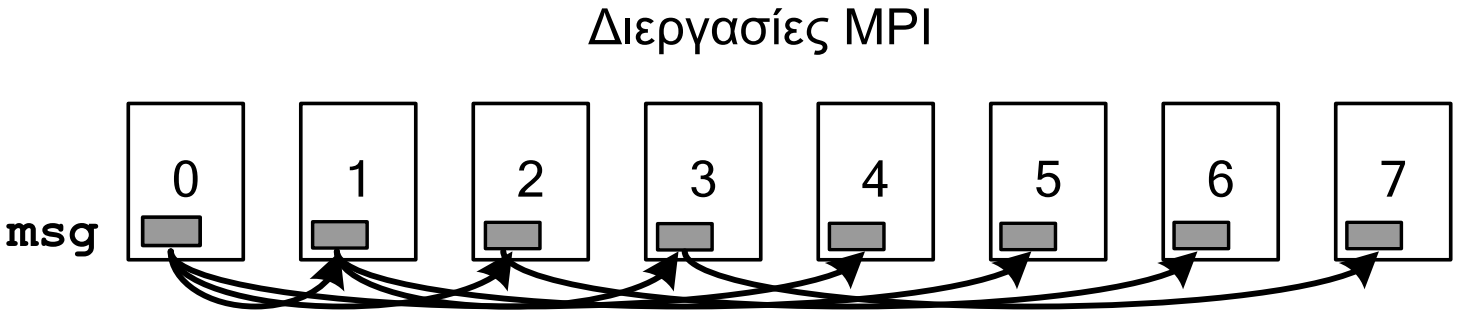
Παράδειγμα: Αποστολή του msg στις διεργασίες 1-7 από τη 0

```
MPI_Bcast(&value,1,MPI_INT,0,MPI_COMM_WORLD);
```



Παράδειγμα: Αποστολή του msg στις διεργασίες 1-7 από τη 0

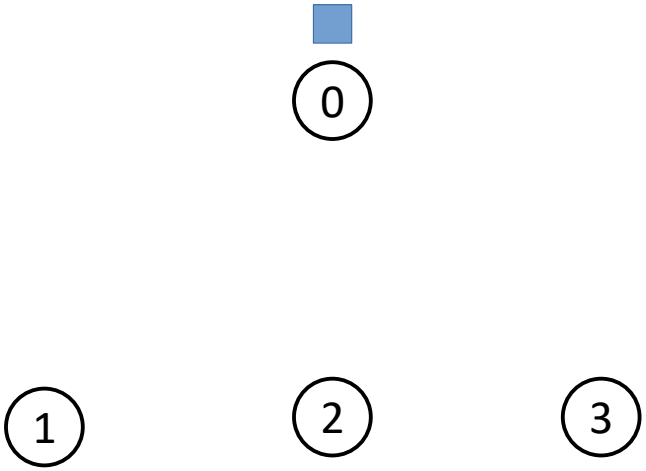
```
MPI_Bcast(&value,1,MPI_INT,0,MPI_COMM_WORLD);
```



Γενικά: Για p διεργασίες έχουμε $\lceil \log_2 p \rceil$ βήματα επικοινωνίας

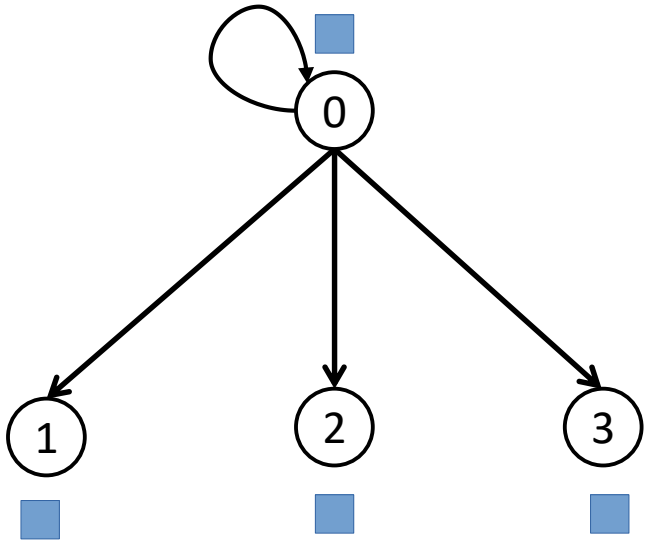
MPI_Bcast

- `MPI_Bcast:`
  - Μία διεργασία στέλνει ένα μήνυμα



MPI_Bcast

- `MPI_Bcast:`
  - Μία διεργασία στέλνει ένα μήνυμα
  - Όλες οι άλλες διεργασίες λαμβάνουν το ίδιο μήνυμα



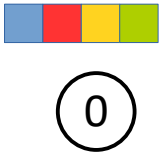
MPI_Bcast

```
int MPI_Bcast(
      void* message,
      int count,
      MPI_Datatype datatype,
      int root,
      MPI_Comm comm)
```

- Αποστολή του `message` από τη διεργασία με rank `root` προς όλες της διεργασίες του communicator `comm`
- Το `message` περιέχει `count` δεδομένα τύπου `datatype`
- Καλείται από όλες τις διεργασίες του `comm`

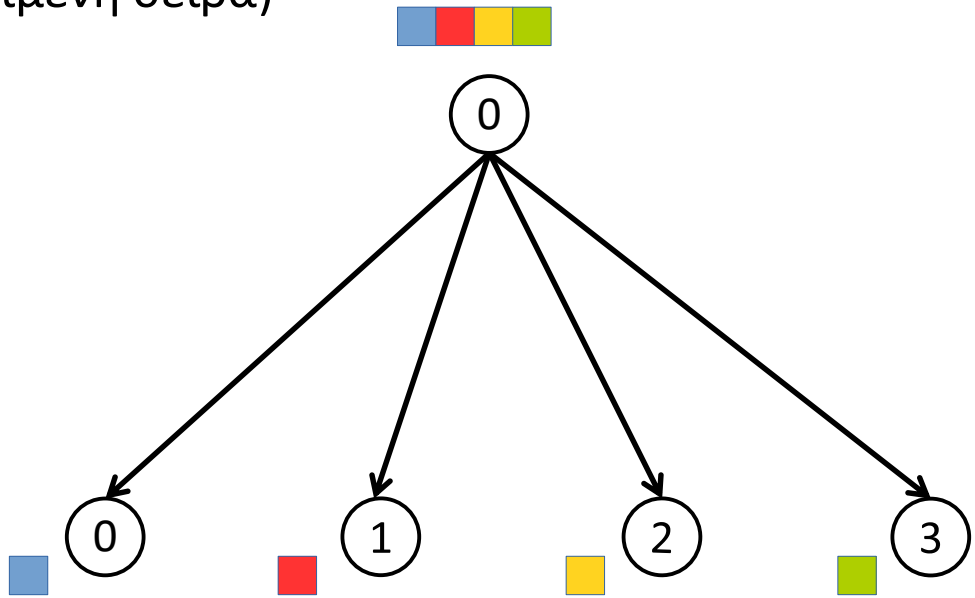
MPI_Scatter

- `MPI_Scatter:`
  - Μία διεργασία στέλνει ένα μήνυμα



MPI_Scatter

- `MPI_Scatter`:
  - Μία διεργασία στέλνει ένα μήνυμα
  - Όλες οι άλλες διεργασίες λαμβάνουν μέρος του μηνύματος (με συγκεκριμένη σειρά)



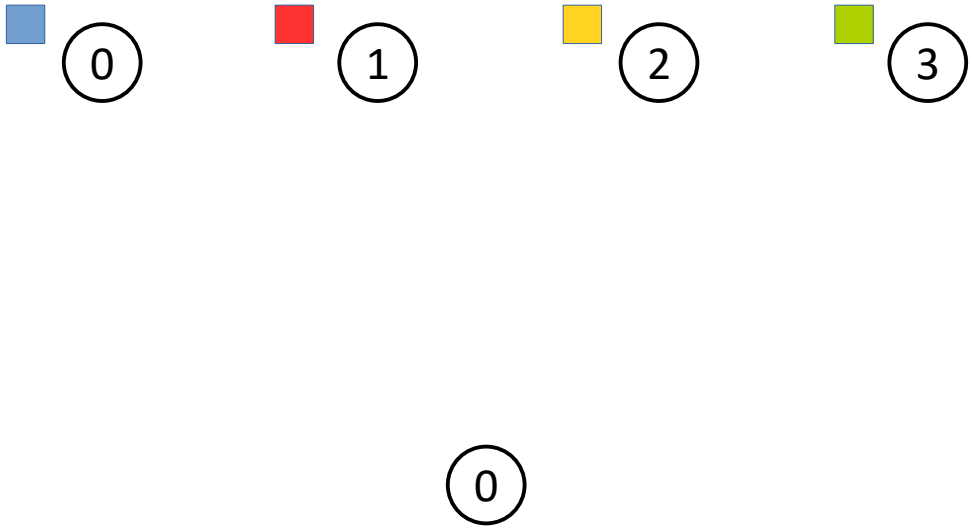
MPI_Scatter

```
int MPI_Scatter(
     const void *sendbuf,
     int sendcount,
     MPI_Datatype sendtype,
     void *recvbuf,
     int recvcount,
     MPI_Datatype recvtype,
     int root,
     MPI_Comm comm)
```

- Η διεργασία `root` μοιράζει τον πίνακα `sendbuf` μήκους `sendcount` μήκους `recvcount`
  - Ο πίνακας `sendbuf` έχει νόημα μόνο στη διεργασία `root`
- Κάθε διεργασία λαμβάνει το κομμάτι του πίνακα που της αντιστοιχεί (κατά αύξουσα σειρά `rank`) και το αποθηκεύει στον πίνακα `recvbuf`
- Αντίστροφη: `MPI_Gather`

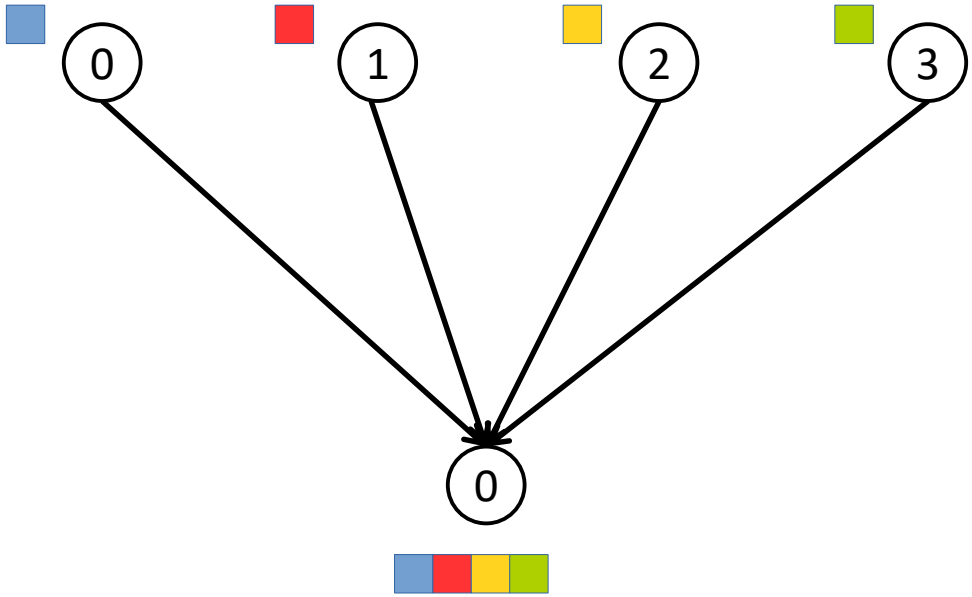
MPI_Gather

- `MPI_Gather:`
  - Όλες οι διεργασίες στέλνουν ένα μήνυμα



MPI_Gather

- `MPI_Gather`:
  - Όλες οι διεργασίες στέλνουν ένα μήνυμα
  - Μία διεργασία λαμβάνει όλα τα μηνύματα και τα συνενώνει



MPI_Gather

```
int MPI_Gather(
     void* sendbuf,
     int sendcnt,
     MPI_Datatype sendtype,
     void* recvbuf,
     int recvcount,
     MPI_Datatype recvtype,
     int root,
     MPI_Comm comm)
```

- Συνενώνονται στη διεργασία `root` οι πίνακες `sendbuf` των υπολοιπών (κατά αύξουσα σειρά `rank`)
- Το αποτέλεσμα αποθηκεύεται στον πίνακα `recvbuf`, ο οποίος έχει νόημα μόνο στη διεργασία `root`
- Αντίστροφη: `MPI_Scatter`

Χειρισμός πινάκων στο MPI: Παράδειγμα scatter/gather

```
/* Εφαρμογή της συνάρτησης foo σε όλα τα στοιχεία ενός πίνακα */

int main(int argc; char **argv;)
{
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    ...

    //Η διεργασία 0 έχει αποθηκεύσει n integers σε έναν πίνακα a
    //Μοιράζει στις άλλες διεργασίες τον πίνακα a (έστω ότι n % size = 0)

    MPI_Scatter(a, n, MPI_INT, a_local, n/size, 0, MPI_COMM_WORLD);

    //Όλες οι διεργασίες έχουν ένα τμήμα του αρχικού πίνακα a μήκους n/size στον a_local

    for (i = 0 ; i < n/size ; i++)
        a_local[i] = foo(a_local[i]);

    //Η διεργασία 0 «μαζεύει» τους πίνακες a_local στον πίνακα a
    MPI_Gather(a_local, n/size, MPI_INT, a, n, MPI_INT, 0, MPI_COMM_WORLD);

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

MPI_Scatterv / MPI_Gatherv

Όταν τα δεδομένα που πρέπει να ανταλλαγούν δεν έχουν το ίδιο μέγεθος σε κάθε διεργασία:

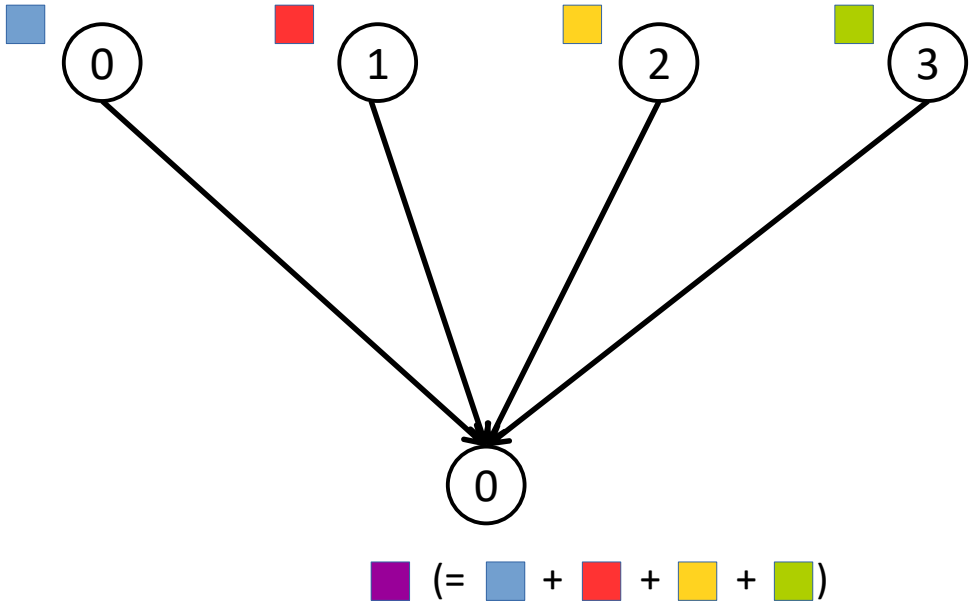
```
int MPI_Scatterv(const void *sendbuf, const int *sendcounts,
                 const int *displs, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm)

int MPI_Gatherv(const void *sendbuf, int sendcount,
                MPI_Datatype sendtype, void *recvbuf,
                const int *recvcounts, const int *displs,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_Reduce

- `MPI_Reduce:`
  - Όλες οι διεργασίες στέλνουν ένα μήνυμα

MPI_Reduce

- `MPI_Reduce:`
  - Όλες οι διεργασίες στέλνουν ένα μήνυμα
  - Μία διεργασία λαμβάνει όλα τα μηνύματα και εφαρμόζει έναν τελεστή



MPI_Op=MPI_SUM

MPI_Reduce

```
int MPI_Reduce(
    void* operand,
    void* result,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm comm)
```

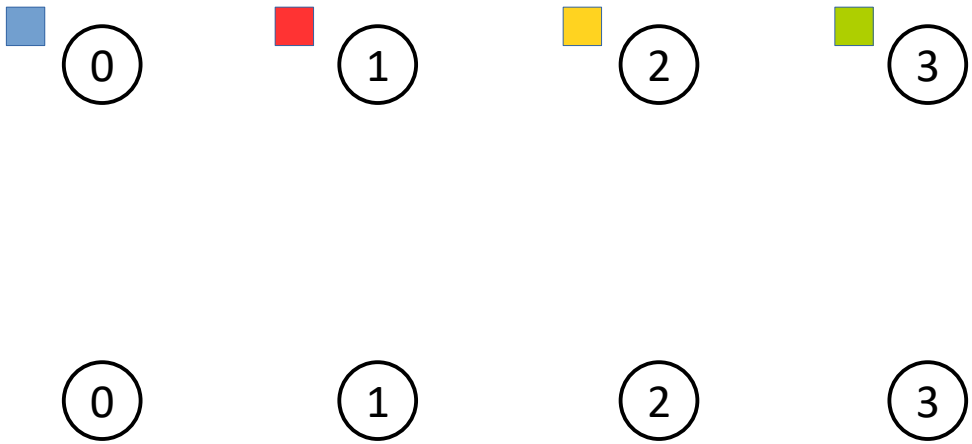
- Τα δεδομένα `operand` συνδυάζονται με εφαρμογή του τελεστή `op`, και το αποτέλεσμα αποθηκεύεται στη διεργασία `root` στο `result`
- Πρέπει να κληθεί από όλες τις διεργασίες του `comm`
- MPI_Op: `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD` κλπ.
- Αντίστοιχα και `MPI_Allreduce`

Χειρισμός πινάκων στο MPI: Παράδειγμα reduce

```
/* Εύρεση του αθροίσματος των στοιχείων ενός πίνακα */

int main(int argc; char **argv;)
{
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    ...

    //Η διεργασία 0 έχει αποθηκεύσει n integers σε έναν πίνακα a
    //Μοιράζει στις άλλες διεργασίες τον πίνακα a

    MPI_Scatter(a, n, MPI_INT, a_local, n/size, 0, MPI_COMM_WORLD);

    //Όλες οι διεργασίες έχουν ένα τμήμα του αρχικού πίνακα a μήκους n/size στον a_local

    for (i = 0 ; i < n/size ; i++)
        sum = sum + a_local[i];

    //Η διεργασία 0 τα τοπικά αθροίσματα sum στο ολικό άθροισμα global_sum
    MPI_Reduce(&sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

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

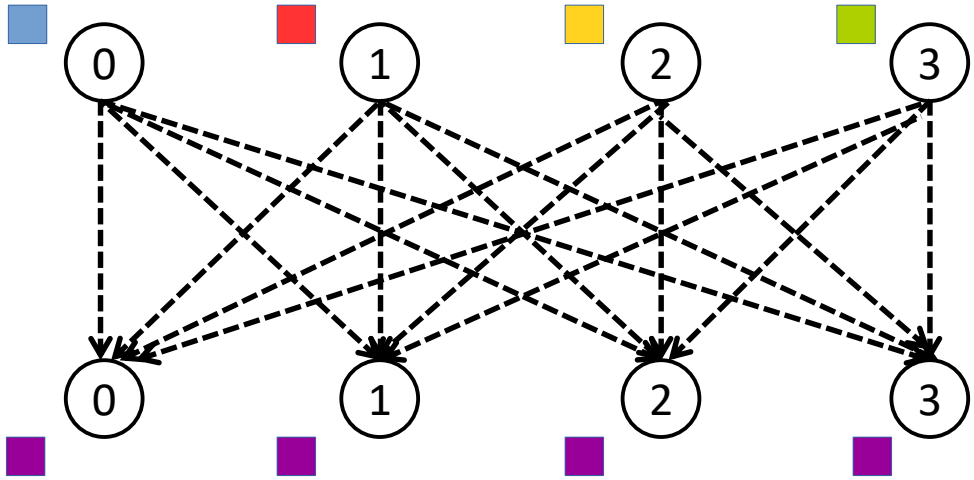
MPI_Allreduce

- `MPI_Allreduce`:
  - Όλες οι διεργασίες στέλνουν ένα μήνυμα



MPI_Allreduce

- `MPI_Allreduce`:
  - Όλες οι διεργασίες στέλνουν ένα μήνυμα
  - Όλες οι διεργασίες λαμβάνουν όλα τα μηνύματα και εφαρμόζουν έναν τελεστή



MPI_Op=MPI_SUM

MPI_Allreduce

```
int MPI_Allreduce(
    void* operand,
    void* result,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm comm)
```

- Τα δεδομένα `operand` συνδυάζονται με εφαρμογή του τελεστή `op`, και το αποτέλεσμα αποθηκεύεται σε όλες τις διεργασίες στο `result`
- Πρέπει να κληθεί από όλες τις διεργασίες του `comm`
- MPI_Op: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD` κλπ.
- Κοστοβόρα επικοινωνία!
- Αντίστοιχα ορίζεται και MPI_Allgather

MPI_Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

- Συγχρονίζονται όλες οι διεργασίες στον communicator `comm`
- Περιορίζει την παραλληλία

Αποστολή μηνυμάτων και συνθήκη επιτυχίας

- Το `MPI_Send` είναι blocking συνάρτηση
  - Σύμφωνα με το πρότυπο, επιστρέφει μόνο όταν είναι ασφαλές να ξαναγραφεί ο buffer από όπου στάλθηκαν τα δεδομένα
  - Δεν εξασφαλίζει την επιτυχία της αποστολής!
  - Υπάρχουν τρεις εναλλακτικοί τρόποι αποστολής μηνυμάτων
    - Με σαφέστερα ορισμένες συνθήκες επιτυχίας/αποτυχίας

Synchronous-Buffered-Ready Send

- Αναφέρονται σε λειτουργία αποστολής, διαφοροποιούνται ως προς λειτουργία λήψης
- Υπάρχουν τόσο σε blocking, όσο και σε non-blocking μορφή
- Το απλό MPI_Send μπορεί να είναι synchronous ή buffered
  - Εξαρτάται από την υλοποίηση και το μέγεθος του μηνύματος

Synchronous-Buffered-Ready

```
int MPI_Ssend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

- Επιτυγχάνει μόνο όταν πάρει επιβεβαίωση λήψης από δέκτη - ασφαλές

```
int MPI_Bsend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

- Επιτυγχάνει όταν αντιγραφεί το μήνυμα σε system buffer για μελλοντική μετάδοση – σφάλμα σε έλλειψη πόρων

```
int MPI_Rsend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

- Επιτυγχάνει μόνο αν έχει προηγηθεί αντίστοιχο receive από το δέκτη – αβέβαιο

Synchronous-Buffered-Ready

MPI_Bsend	MPI_Ssend	MPI_Rsend
Τοπικό	Μη τοπικό	Τοπικό
2 αντιγραφές στη μνήμη	1 αντιγραφή στη μνήμη	1 αντιγραφή στη μνήμη
Αποτυγχάνει ελλείψει πόρων	Δεν αποτυγχάνει ελλείψει πόρων	Δεν αποτυγχάνει ελλείψει πόρων
Δεν αποτυγχάνει αν δεν έχει προηγηθεί λήψη	Δεν αποτυγχάνει αν δεν έχει προηγηθεί λήψη	Αποτυγχάνει αν δεν έχει προηγηθεί λήψη

Σύγχρονη και ασύγχρονη επικοινωνία

- Τα `MPI_Send / MPI_Recv` είναι blocking συναρτήσεις
  - Το `MPI_Send` επιστρέφει μόνο όταν είναι ασφαλές να ξαναγραφεί ο buffer από όπου στάλθηκαν τα δεδομένα
  - Το `MPI_Recv` επιστρέφει μόνο όταν ολόκληρο το μήνυμα έχει παραληφθεί – αποθηκευτεί στον buffer

- Εναλλακτικά, μπορεί κανείς να χρησιμοποιήσει τα `MPI_Isend / MPI_Irecv`
  - Non-blocking συναρτήσεις
  - Επιστρέφουν άμεσα (*I=immediate*) ένα request στο χρήστη
  - Ο χρήστης μπορεί να ελέγξει την κατάσταση αποστολής/λήψης με `MPI_Test` στο request
  - Ο χρήστης μπορεί να περιμένει την ολοκλήρωση της επικοινωνίας με `MPI_Wait` στο request
  - Από τη στιγμή που επιστρέφει η συνάρτηση μέχρι και την κλήση σε συνάρτηση Test/Wait, ο επεξεργαστής είναι ελεύθερος να πραγματοποιήσει οποιαδήποτε άλλη εργασία

MPI_Isend

```
int MPI_Isend(
        void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm,
        MPI_Request * request)
```

- Αποστολή μηνύματος `buf` από καλούσα διεργασία σε διεργασία με rank `dest`
- Ο πίνακας `buf` έχει `count` στοιχεία τύπου `datatype`
- Όταν η συνάρτηση επιστρέφει, το μήνυμα μπορεί να είναι:
  - Αποθηκευμένο σε εσωτερικό buffer
  - Υπό αποστολή

Παράδειγμα:

```
int message[20],dest=1,tag=55;
MPI_Request request;
MPI_Isend(message, 20, MPI_INT, dest, tag, MPI_COMM_WORLD, &request);
//Απαιτείται έλεγχος του request για εξασφάλιση ολοκλήρωσης αποστολής
```

MPI_Irecv

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

- Λήψη μηνύματος από διεργασία με rank `source` και αποθήκευση σε `buf`
- Λαμβάνονται το πολύ `count` δεδομένα τύπου `datatype` (ακριβής αριθμός με `MPI_Get_count`)
- Wildcards: `MPI_ANY_SOURCE, MPI_ANY_TAG`
- Όταν η συνάρτηση επιστρέφει, το μήνυμα μπορεί να είναι:
  - Σε λήψη (in transit)
  - Αποθηκευμένο σε εσωτερικό buffer στην πλευρά του παραλήπτη
  - ...ή να μην έχει ξεκινήσει η αποστολή του

Παράδειγμα:

```
int message[50],source=0,tag=55;
MPI_Request request;
MPI_Irecv(message, 50, MPI_INT, source, tag MPI_COMM_WORLD,&request);

 //Απαιτείται έλεγχος του request για εξασφάλιση ολοκλήρωσης λήψης
```

Non-blocking communication

- Άμεση επιστροφή
- Δεν είναι ασφαλές να επαναχρησιμοποιηθούν οι buffers επικοινωνίας πριν ελεγχθεί η επιτυχία

- Δύο δυνατότητες για έλεγχο επιτυχίας της επικοινωνίας:
  - `int MPI_Test (MPI_Request* request,int* flag, MPI_Status* status)`
  - `int MPI_Wait (MPI_Request* request,MPI_Status* status)`
    - `int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])`
    - `Int MPI_Waitany(int count, MPI_Request array_of_requests[], int *indx, MPI_Status *status)`

- Δυνατότητα ελέγχου για εισερχόμενα μηνύματα, χωρίς να γίνει λήψη (progress report):
  - `int MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status)`

Non-blocking communication

- Κάθε blocking συνάρτηση έχει την αντίστοιχη non-blocking:
  - MPI_Isend (για MPI_Send)
  - MPI_Issend (για MPI_Ssend)
  - MPI_Ibsend (για MPI_Bsend)
  - MPI_Irsend (για MPI_Rsend)
  - MPI_Irecv (για MPI_Recv)

Επικάλυψη υπολογισμών - επικοινωνίας

- **Blocking**

```
MPI_Recv();
MPI_Send();
Compute();
```

- **Non-blocking**

```
MPI_Irecv();
MPI_Isend();
Compute();
Waitall();
```

Αποφυγή deadlocks

- **Blocking (πιθανό deadlock!)**

Rank 0	Rank 1
`MPI_Send(msg,Rank1);`	`MPI_Send(msg,Rank0);`
`MPI_Recv(msg,Rank1);`	`MPI_Recv(msg,Rank0);`
`Compute();`	`Compute();`

- **Non-blocking**

Rank 0	Rank 1
`MPI_Isend(msg,Rank1);`	`MPI_Isend(msg,Rank0);`
`MPI_Irecv(msg,Rank1);`	`MPI_Irecv(msg,Rank0);`
`Waitall();`	`Waitall();`
`Compute();`	`Compute();`

Τύποι δεδομένων MPI

- MPI_CHAR: 8-bit χαρακτήρας
- MPI_DOUBLE: 64-bit κινητής υποδιαστολής
- MPI_FLOAT: 32-bit κινητής υποδιαστολής
- MPI_INT: 32-bit ακέραιος
- MPI_LONG: 32-bit ακέραιος
- MPI_LONG_DOUBLE: 64-bit κινητής υποδιαστολής MPI_LONG_LONG: 64-bit ακέραιος
- MPI_LONG_LONG_INT: 64-bit ακέραιος
- MPI_SHORT: 16-bit ακέραιος
- MPI_SIGNED_CHAR: 8-bit προσημασμένος χαρακτήρας
- MPI_UNSIGNED: 32-bit απρόσημος ακέραιος MPI_UNSIGNED_CHAR: 8-bit απρόσημος χαρακτήρας MPI_UNSIGNED_LONG: 32-bit απρόσημος ακέραιος MPI_UNSIGNED_LONG_LONG: 64-bit απρόσημος ακέραιος MPI_UNSIGNED_SHORT: 16-bit απρόσημος ακέραιος
- MPI_WCHAR: 16-bit απρόσημος χαρακτήρας

Τύποι δεδομένων MPI

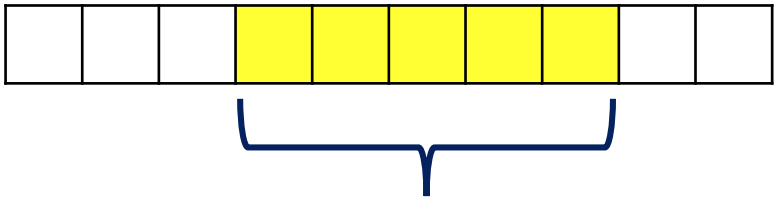
- Ομαδοποίηση δεδομένων επικοινωνίας:
  - Παράμετρος `count` (για ομοιογενή δεδομένα σε συνεχόμενες θέσεις μνήμης)
- `MPI_Type_struct` (derived datatype)
- `MPI_Pack(),MPI_Unpack()` (για ετερογενή δεδομένα)

Τύποι δεδομένων MPI

- Derived datatypes
- Χρησιμοποιούμε τα derived datatypes όταν θέλουμε να ορίσουμε:
  - μη συνεχόμενα στη μνήμη δεδομένα ίδιου τύπου (π.χ. μια στήλη ενός πίνακα)
  - συνεχόμενα στη μνήμη δεδομένα διαφορετικών τύπων (π.χ. ένα struct)
  - μη συνεχόμενα στη μνήμη δεδομένα διαφορετικών τύπων
- Παράδειγμα χρήσης: Πώς μπορώ να στείλω μια στήλη ενός πίνακα;
  - 1. Με μία κλήση send για κάθε στοιχείο
  - 2. Με αντιγραφή της στήλης σε ειδικό buffer
  - 3. Με τη δημιουργία ενός datatype για τη στήλη του πίνακα
- Το 3ο δεν είναι απαραίτητα πιο γρήγορο από το 2ο - εξαρτάται από την υλοποίηση!

MPI_Type_contiguous

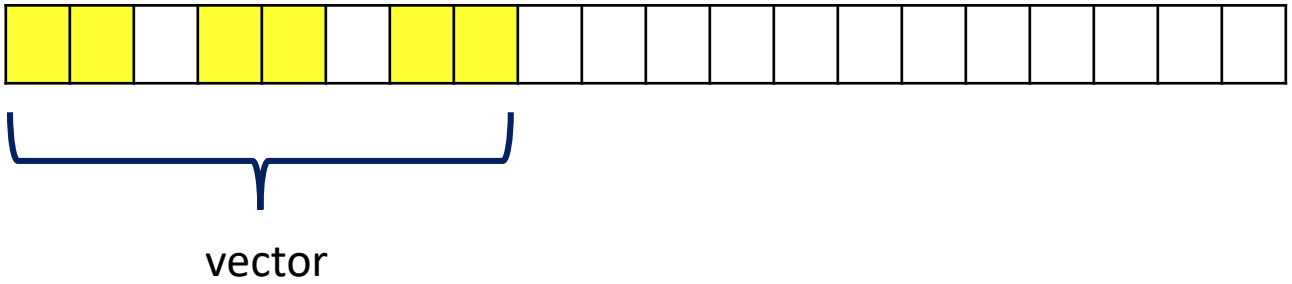
```
int MPI_Type_contiguous(
    int count,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```



- Δημιουργεί ένα νέο datatype `newtype` ενώνοντας `count` συνεχόμενα στη μνήμη δεδομένα τύπου `oldtype`

MPI_Type_vector

```
int MPI_Type_vector(
    int count,
    int blocklength,
    int stride,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```



- Δημιουργεί ένα νέο datatype `newtype` ενώνοντας `blocklength` δεδομένα τύπου `oldtype` από `count` blocks που απέχουν κατά κάποιο `stride`
  - Στο παράδειγμα: blocklength=2, stride=3

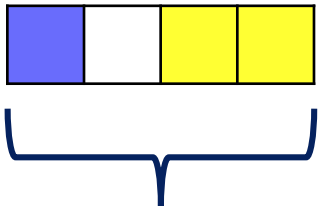
MPI_Type_indexed

```
int MPI_Type_indexed(
    int count,
    int * array_of_blocklengths,
    int *array_of_displacements,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

- Συνενώνει irregular υποσύνολα δεδομένων από πίνακα
- “Ακριβός” τύπος δεδομένων

MPI_Type_create_struct

```
int MPI_Type_create_struct(
     int count,
     int *array_of_blocklengths,
     MPI_Aint *array_of_displacements,
     MPI_Datatype *array_of_types,
     MPI_Datatype *newtype)
```



struct

- Δημιουργεί structs αντίστοιχα με τα structs της C
- Επίσης “ακριβός” τύπος δεδομένων

Χρήση των τύπων δεδομένων MPI

- Κατασκευή του datatype
  - `MPI_Type_contiguous(10, MPI_DOUBLE, &my_datatype)`
- Δέσμευση χώρου για το datatype
  - `MPI_Type_commit(&my_datatype)`
- Χρήση του datatype
  - `MPI_Send(&my_array[100],1,my_datatype,1,55, MPI_COMM_WORLD)`
- Απελευθέρωση του χώρου στη μνήμη
  - `MPI_Type_free(&my_datatype)`

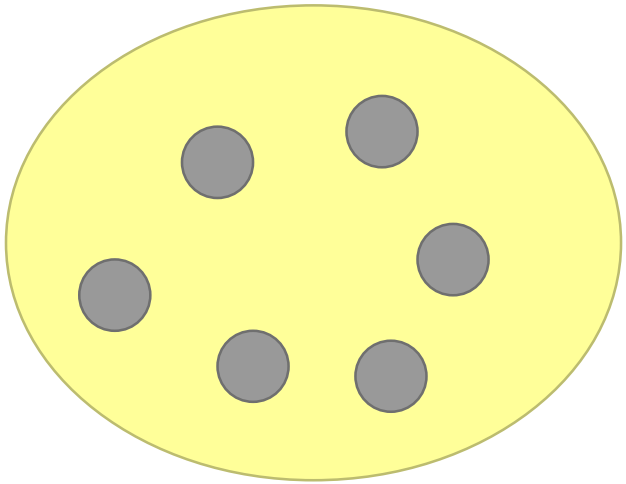
Διεργασίες και communicators

- Ξεκινώντας ένα πρόγραμμα MPI με P διεργασίες, δημιουργείται ένας **communicator** με όνομα `MPI_COMM_WORLD` μεγέθους P διεργασιών
- Σε κάθε διεργασία αποδίδεται ένα μοναδικό **rank** (αναγνωριστικό) στο εύρος 0...P-1, όπου P το συνολικό πλήθος διεργασιών στον συγκεκριμένο communicator
  - *Όχι ακριβώς έτσι...*

- Το MPI ορίζει **groups** διεργασιών
  - Ένα group είναι ένα σύνολο διεργασιών με αυστηρή διάταξη
  - Σε κάθε διεργασία αποδίδεται ένα μοναδικό **rank** (αναγνωριστικό) στο έυρος 0...P-1, όπου P το συνολικό πλήθος διεργασιών στο συγκεκριμένο **group**

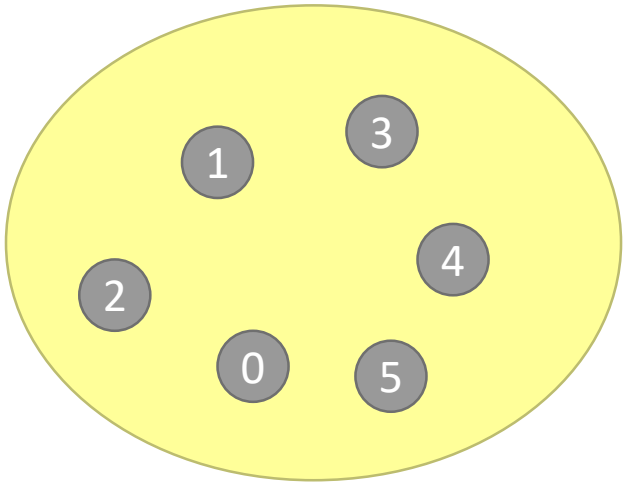
- Το MPI-1 ορίζει τον **communicator** σαν ένα περιβάλλον όπου ένα σύνολο από διεργασίες μπορούν να επικοινωνούν μεταξύ τους
  - Ο `MPI_COMM_WORLD` είναι ο communicator που περιλαμβάνει όλες τις διεργασίες και δημιουργείται κατά το `MPI_Init`
  - Μπορούμε να δημιουργήσουμε οποιονδήποτε communicator για οποιοδήποτε group διεργασιών

MPI Communicators: MPI_Comm_split



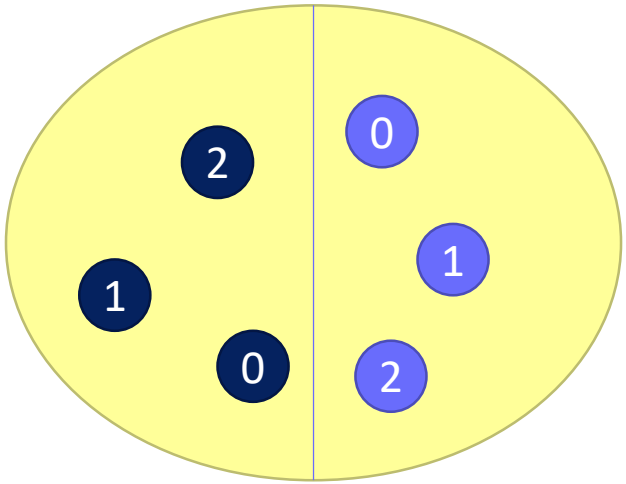
```
MPI_Init(&argc, &argv);
```

MPI Communicators: MPI_Comm_split




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

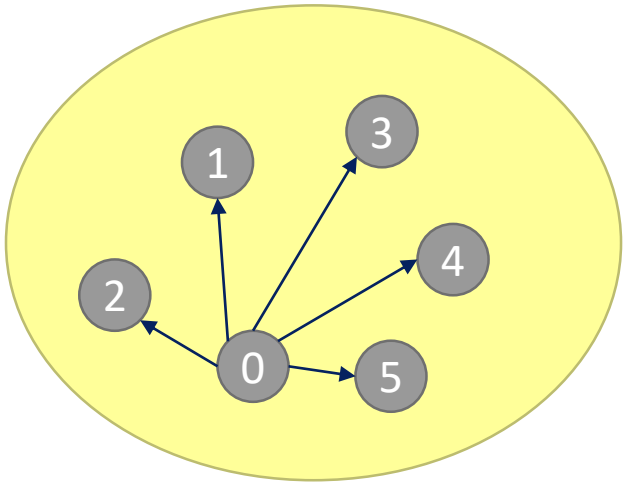
MPI Communicators: MPI_Comm_split




```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
              &rank);
int color= rank/3;
MPI_Comm_split(MPI_COMM_WORLD,
       color, rank,
       split_comm);
MPI_Comm_rank(split_comm,
       &new_rank);
```

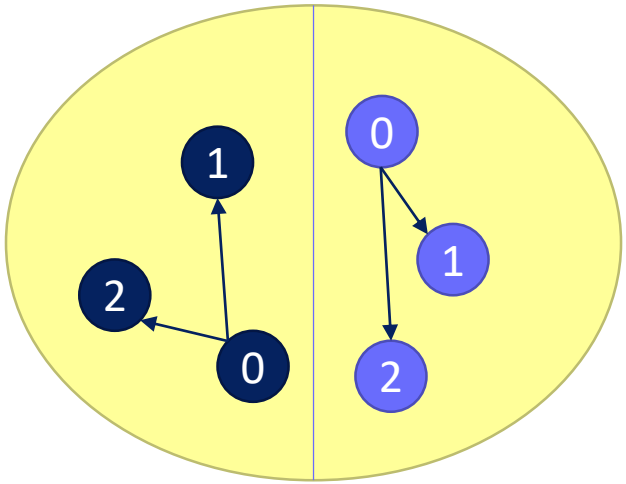
- Ο communicator `MPI_COMM_WORLD` συνεχίζει να υπάρχει και μετά την κλήση `MPI_Comm_split`
- Ο communicator `split_comm` μας βοηθάει να αναφερόμαστε στις διεργασίες 0,1,2 και 3,4,5 ως διαφορετικά groups με νέα ranks (`new_rank`)

MPI Communicators: MPI_Comm_split




```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
              &rank);
int color= rank/3;
MPI_Comm_split(MPI_COMM_WORLD,
       color, rank,
       split_comm);
MPI_Comm_rank(split_comm,
       &new_rank);
...
MPI_Bcast(&a, 1, MPI_INT,
        0, MPI_COMM_WORLD);
```

MPI Communicators: MPI_Comm_split




```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
              &rank);
int color= rank/3;
MPI_Comm_split(MPI_COMM_WORLD,
       color, rank,
       split_comm);
MPI_Comm_rank(split_comm,
       &new_rank);
...
MPI_Bcast(&a, 1, MPI_INT,
        0, MPI_COMM_WORLD);
...
MPI_Bcast(&a, 1, MPI_INT,
       0, split_comm);
```

MPI Communicators: Βασικές λειτουργίες

- `MPI_Comm_split(MPI_Comm comm, int color, int rank, MPI_Comm *newcomm)`
  - Διαχωρισμός των διεργασιών του communicator `comm` με βάση το αναγνωριστικό `color` κάθε διεργασίας στο νέο communicator `newcomm`

- `MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)`
  - Αντιγραφή (duplication) του communicator `comm` στο νέο communicator `newcomm`

- `MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)`
  - Δημιουργία νέου communicator `newcomm` που περιλαμβάνει τις διεργασίες του communicator `comm` που ανήκουν στο group `group`

- `MPI_Comm_free(MPI_Comm *comm)`
  - Διαγραφή του communicator `comm`

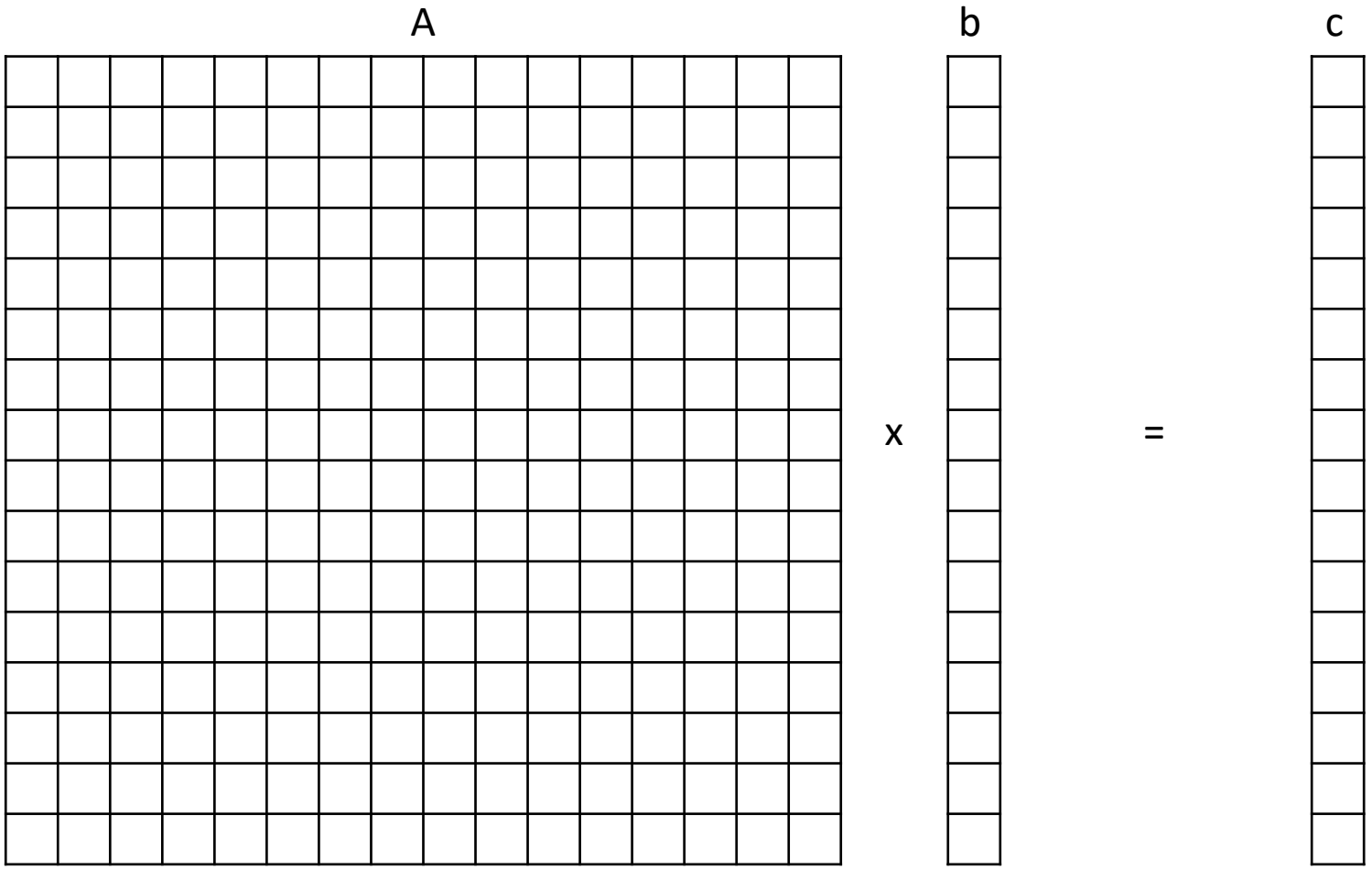
MPI Groups

- Τα groups είναι ομάδες διεργασιών με αυστηρή διάταξη
  - Κάθε διεργασία έχει ένα συγκεκριμένο rank σε κάθε group στο οποίο ανήκει
  - Μπορούμε να δημιουργήσουμε περισσότερους από έναν communicators για κάθε group

- `MPI_Group_create(MPI_Comm comm, MPI_Group group)`
  - Δημιουργία του group `group` που περιλαμβάνει όλες τις διεργασίες του communicator `comm`
    - Χρησιμοποιείται για τη δημιουργία ενός group που περιλαμβάνει όλες τις διεργασίες του `MPI_COMM_WORLD`
- `MPI_Group_incl(MPI_Group group, int n, int * ranks, MPI_Group * newgroup)`
  - Προσθήκη `n` διεργασιών με αναγνωριστικά `ranks` από το υπάρχον group `group` στο νέο group `newgroup`
- `MPI_Group_excl(MPI_Group group, int n, int * ranks, MPI_Group * newgroup)`
  - Αφαίρεση `n` διεργασιών με αναγωριστικά `ranks` από το υπάρχον group `group` στο νέο group `newgroup`
- `MPI_Group_union, MPI_Group_intersection, MPI_Group_difference...`

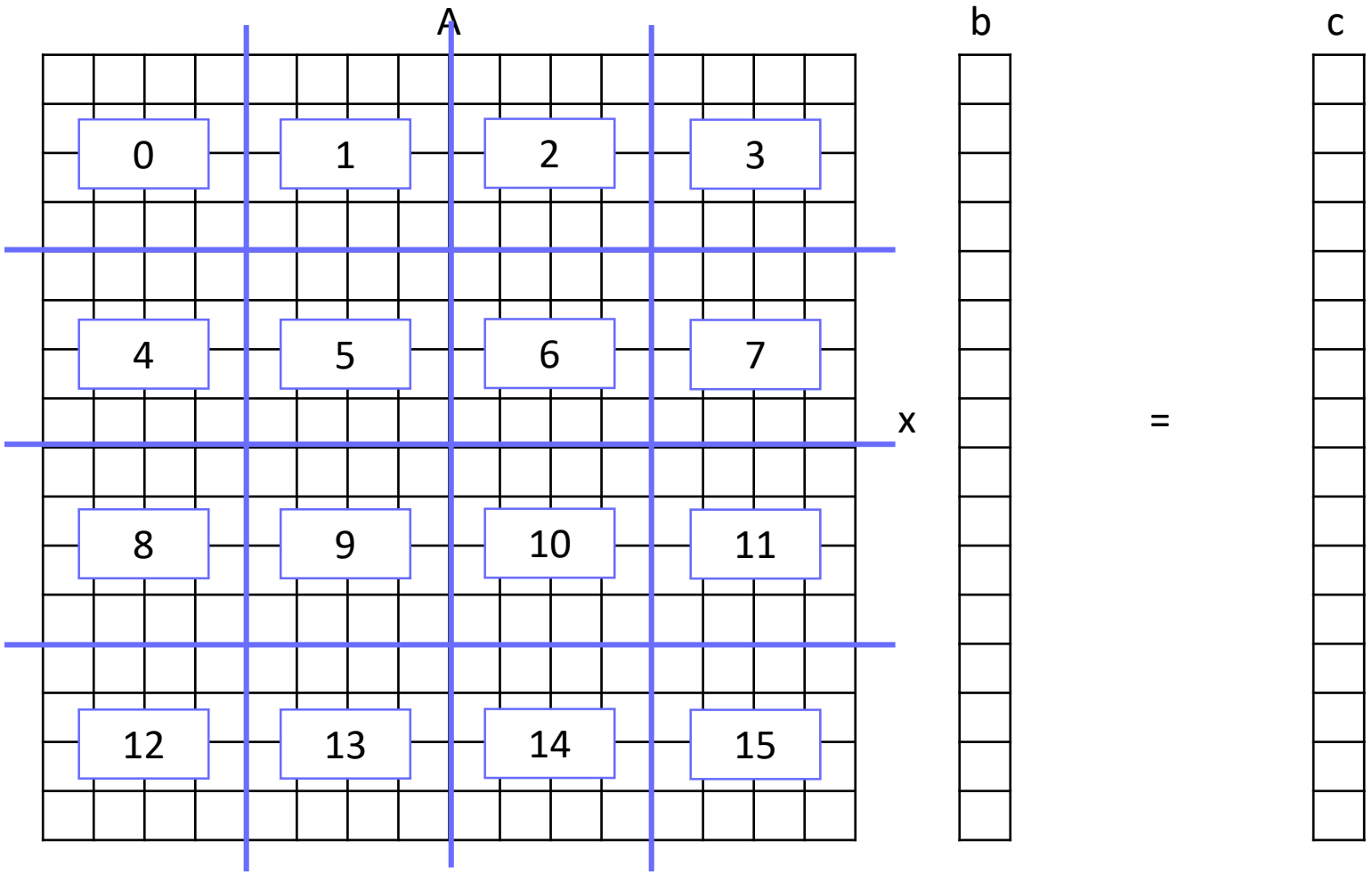
Παράδειγμα χρήσης MPI Groups/Communicators

- *Παράδειγμα*: Πολλαπλασιασμός πίνακα με διάνυσμα



Παράδειγμα χρήσης MPI Groups/Communicators

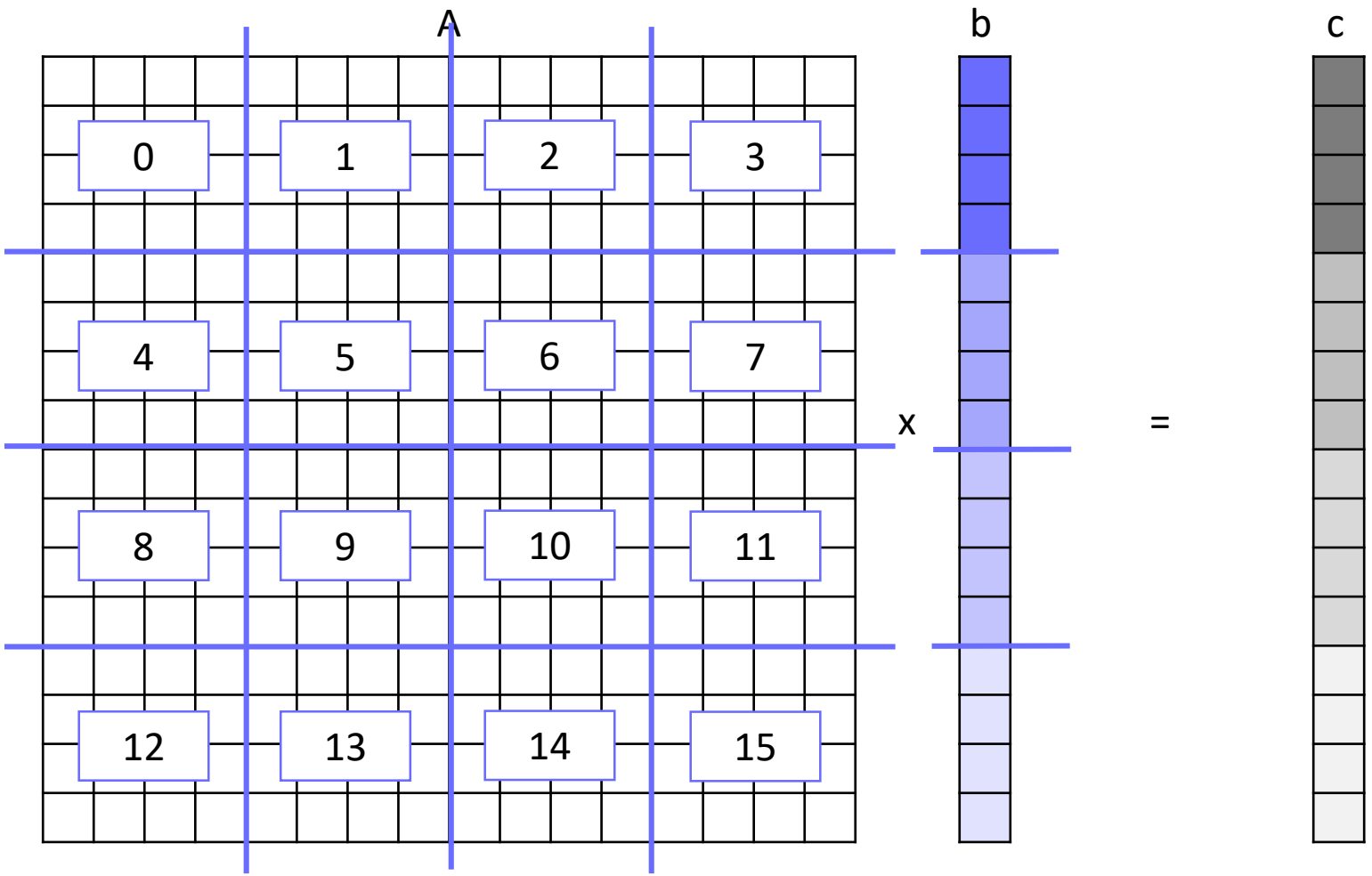
- *Παράδειγμα*: Πολλαπλασιασμός πίνακα με διάνυσμα



National Technical University of Athens
CSLab

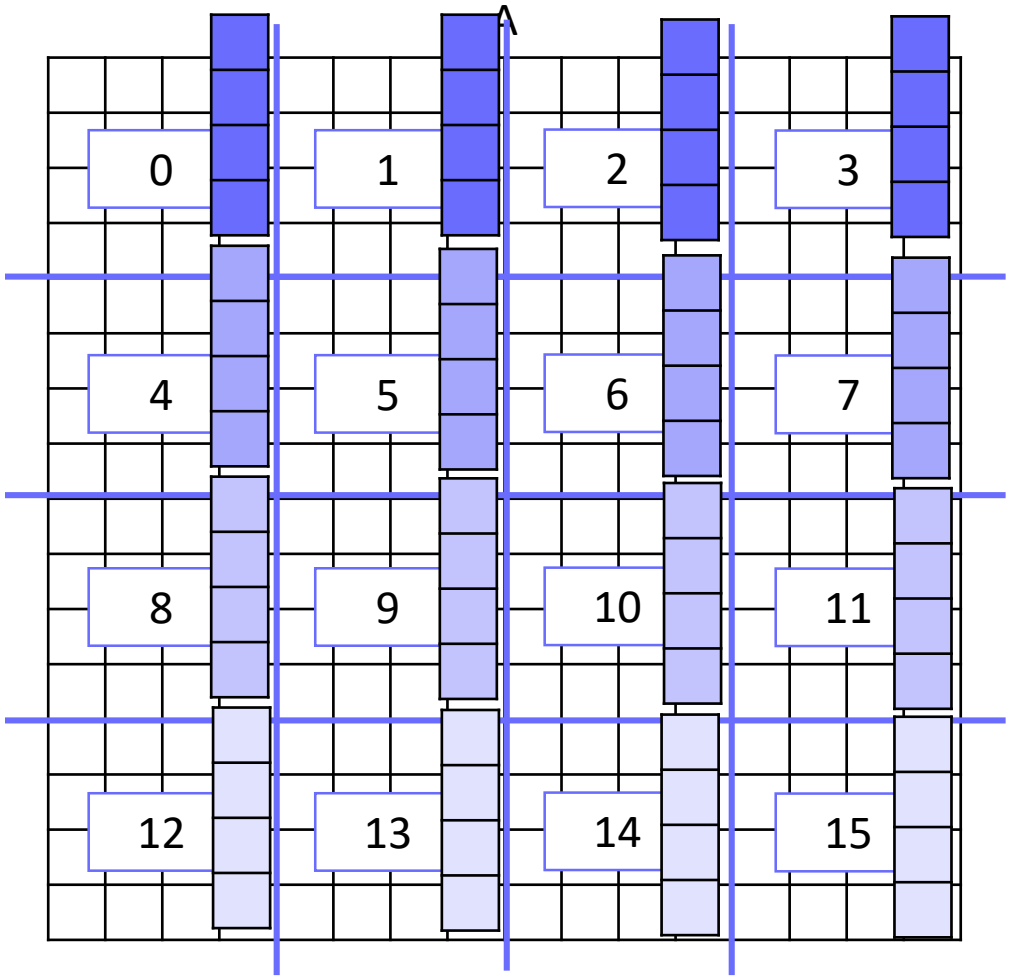
Παράδειγμα χρήσης MPI Groups/Communicators

- *Παράδειγμα*: Πολλαπλασιασμός πίνακα με διάνυσμα



Παράδειγμα χρήσης MPI Groups/Communicators

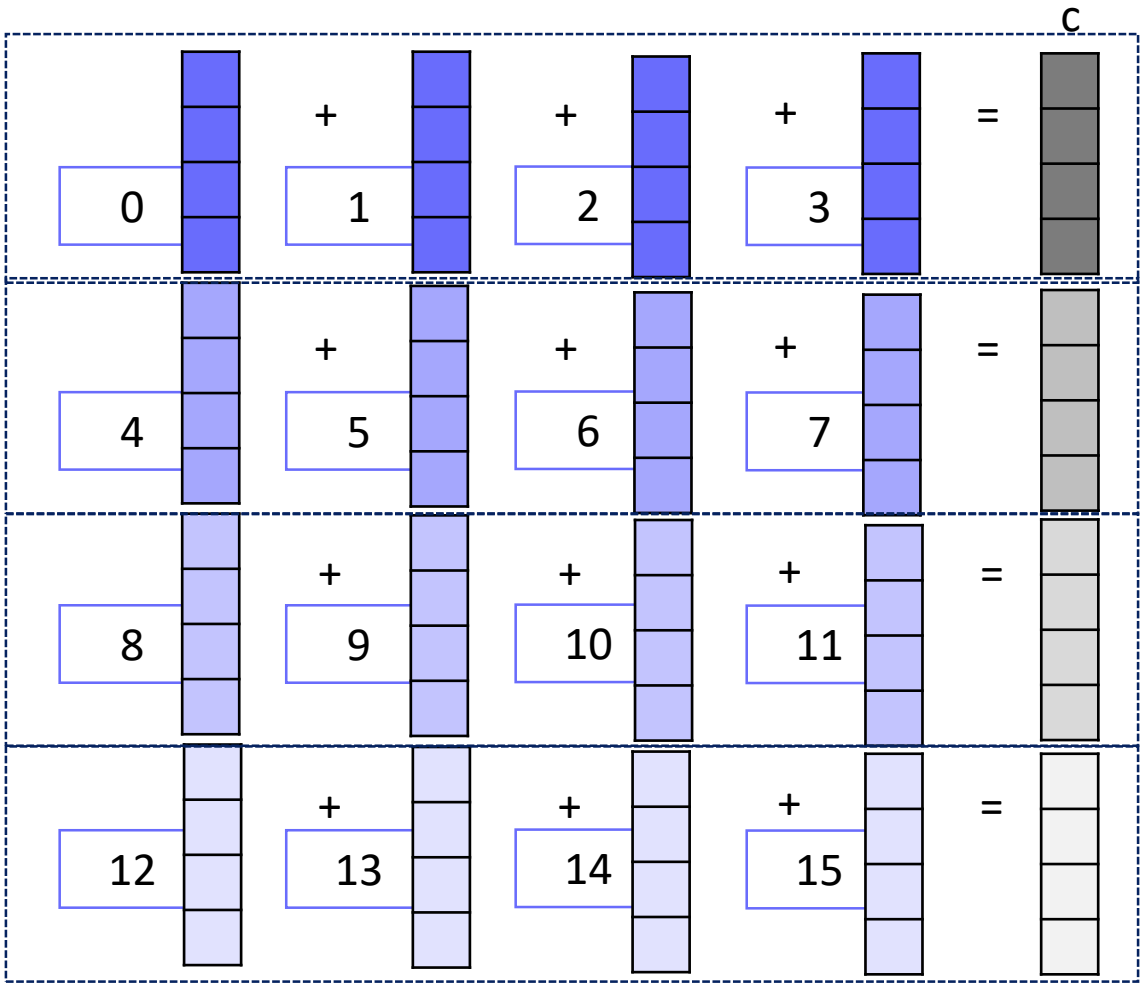
- *Παράδειγμα*: Πολλαπλασιασμός πίνακα με διάνυσμα



- Κάθε διεργασία αναλαμβάνει ένα τμήμα του πίνακα A κι ένα μέρος του διανύσματος b
- Υπολογίζει τον πολλαπλασιασμό αυτού του μέρους του πίνακα με το μέρος του διανύσματος b

Παράδειγμα χρήσης MPI Groups/Communicators

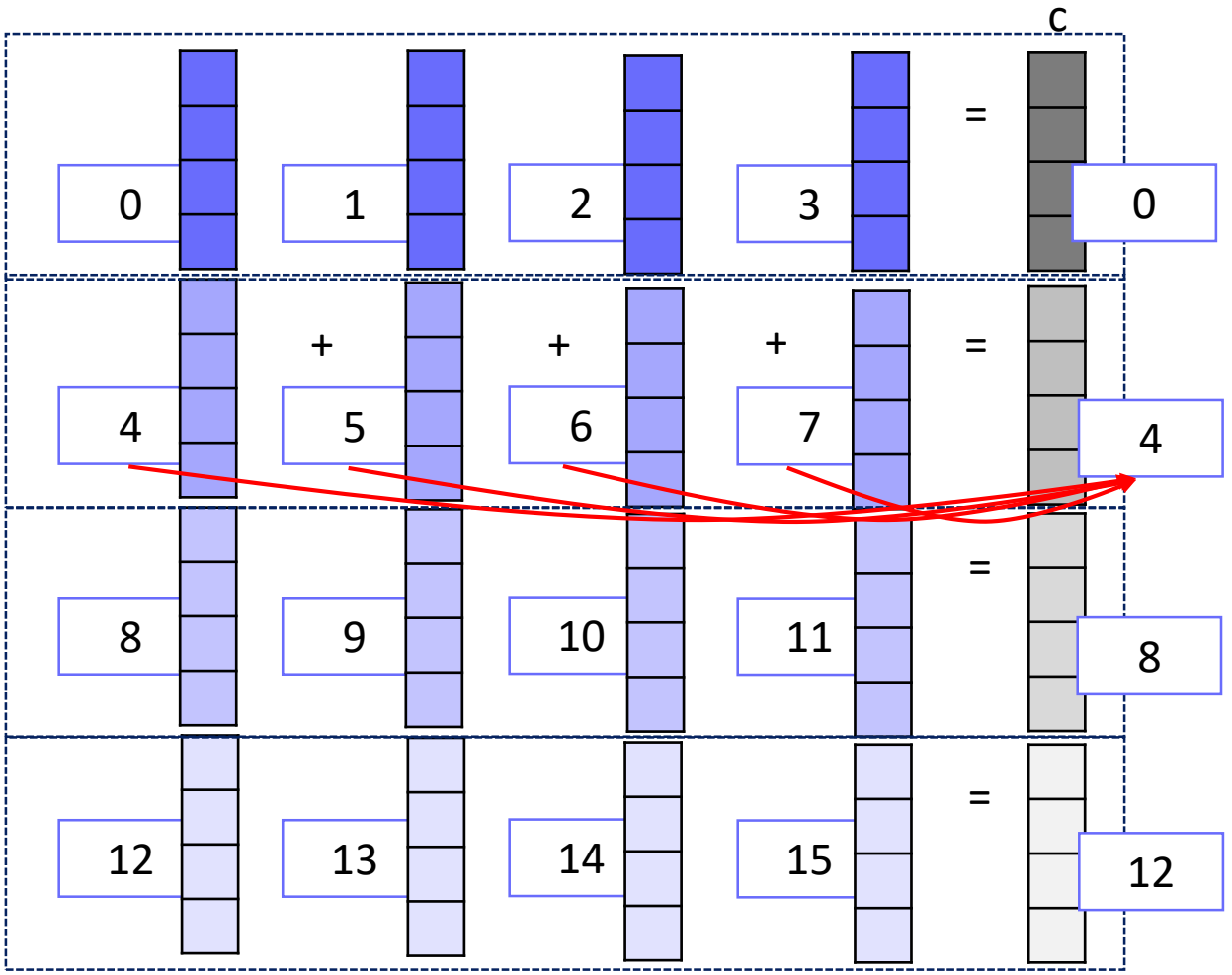
- *Παράδειγμα*: Πολλαπλασιασμός πίνακα με διάνυσμα



- Τα μερικά γινόμενα πρέπει να προστεθούν για να προκύψει το αντίστοιχο μέρος του διανύσματος c

Παράδειγμα χρήσης MPI Groups/Communicators

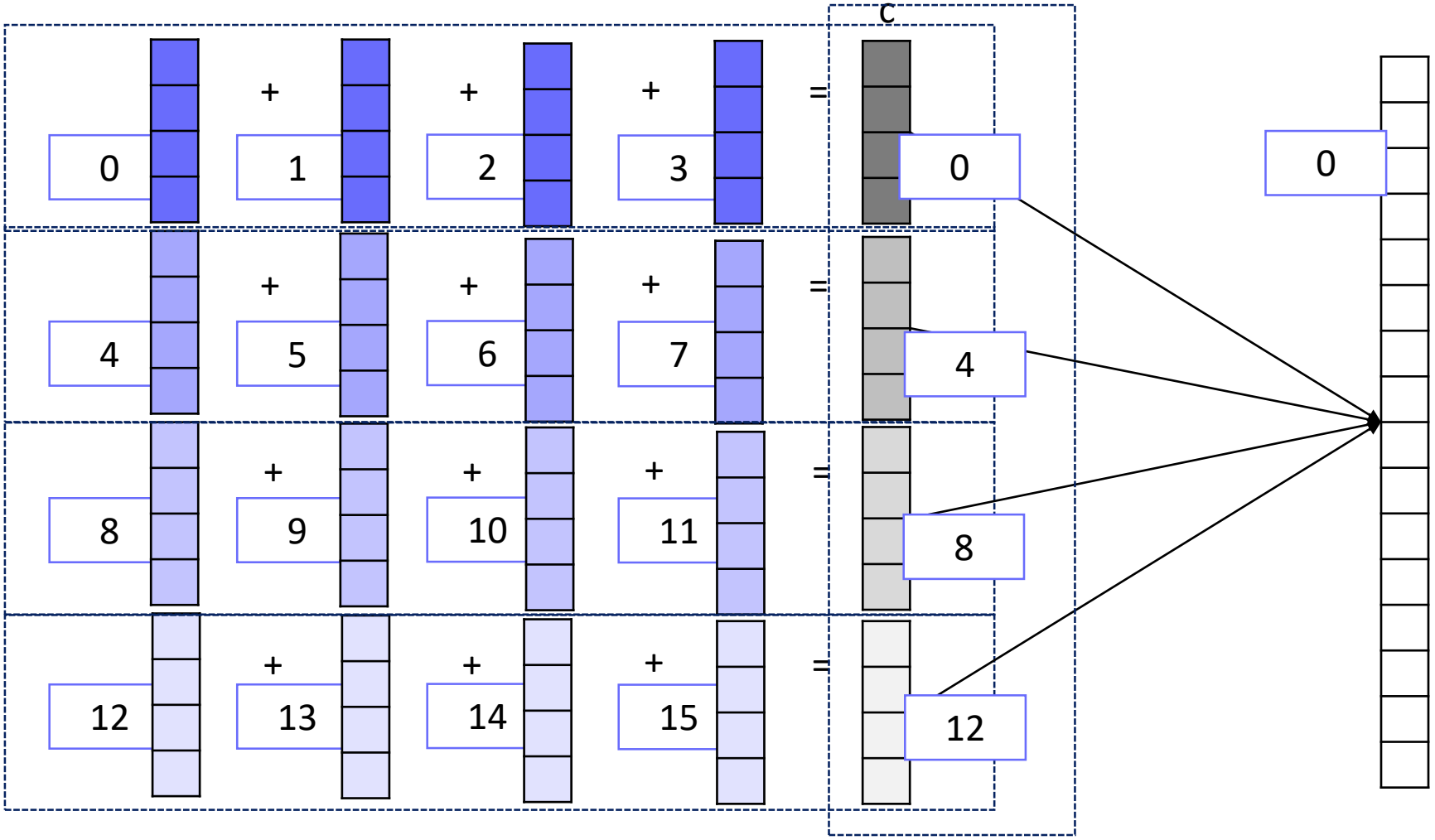
- *Παράδειγμα*: Πολλαπλασιασμός πίνακα με διάνυσμα



- Φτιάχνουμε groups που περιλαμβάνουν διεργασίες στην ίδια γραμμή
- Κάνουμε «μερικά» reductions

Παράδειγμα χρήσης MPI Communicators

- *Παράδειγμα*: Πολλαπλασιασμός πίνακα με διάνυσμα



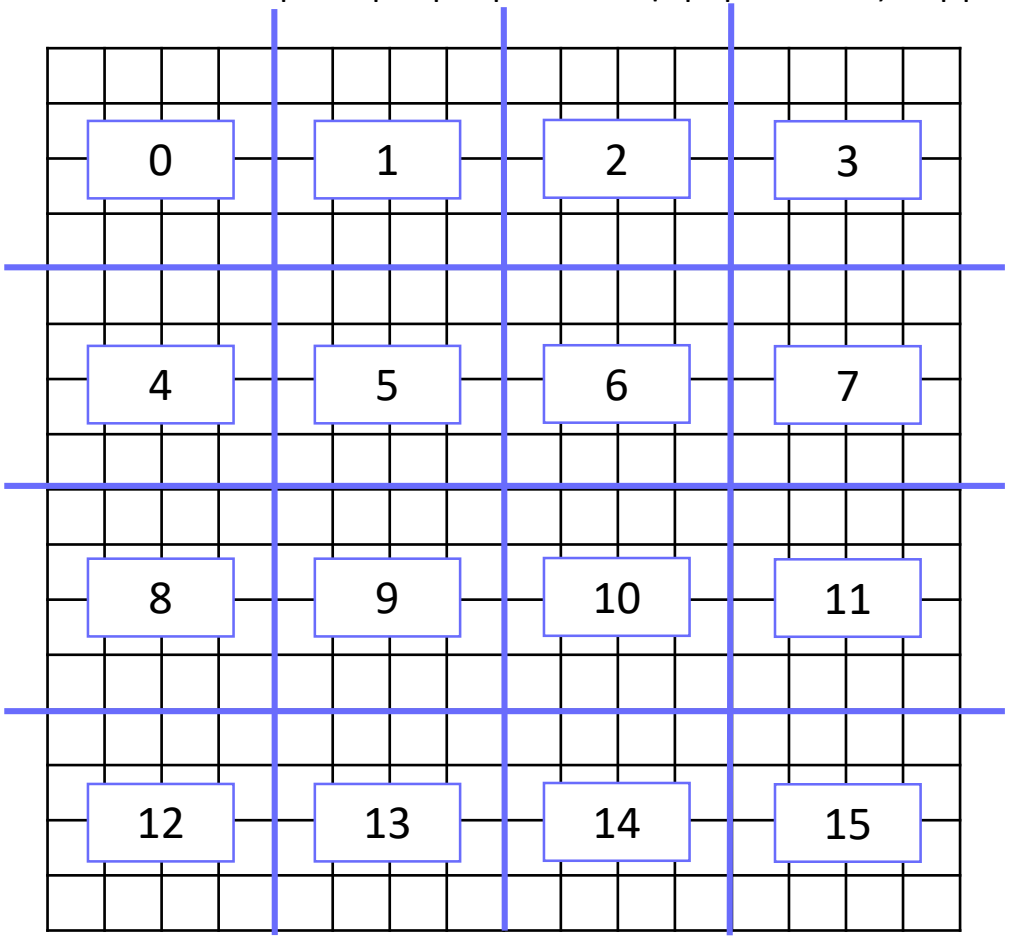
Εικονικές τοπολογίες

- Το MPI παρέχει ρουτίνες για τη δημιουργία νέων communicators όπου η διάταξη των διεργασιών εκφράζει πιο φυσικά την επικοινωνία μεταξύ των διεργασιών

- Δύο βασικές τοπολογίες:
  - Καρτεσιανή τοπολογία
  - Τοπολογία γράφου (γενική)

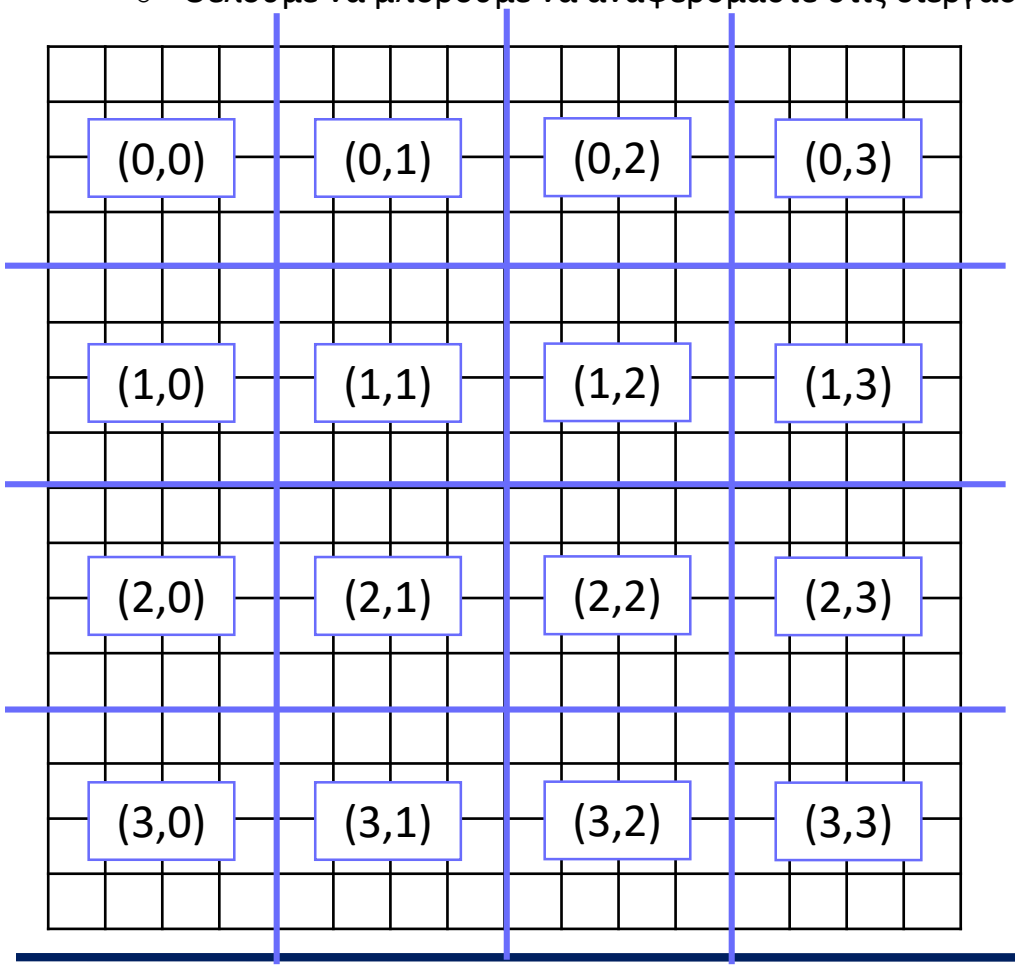
Καρτεσιανή τοπολογία στο MPI

- *Παράδειγμα:* Μοιράζουμε ένα 2Δ-χωρίο σε 2Δ-υποχωρία στις διεργασίες του MPI
  - Οι διεργασίες δημιουργούν ένα εικονικό 2Δ-πλέγμα
  - Θέλουμε να μπορούμε να αναφερόμαστε στις διεργασίες με βάση τη θέση τους στο πλέγμα



Καρτεσιανή τοπολογία στο MPI

- *Παράδειγμα:* Μοιράζουμε ένα 2Δ-χωρίο σε 2Δ-υποχωρία στις διεργασίες του MPI
  - Οι διεργασίες δημιουργούν ένα εικονικό 2Δ-πλέγμα
  - Θέλουμε να μπορούμε να αναφερόμαστε στις διεργασίες με βάση τη θέση τους στο πλέγμα



Η εικονική καρτεσιανή τοπολογία του MPI μπορεί να βοηθήσει

Καρτεσιανή τοπολογία στο MPI

```
MPI_Cart_create(
      MPI_Comm comm,
      int ndim, int dims[],
      int qperiodic[], int qreorder,
      MPI_Comm * new_comm)
```

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

- Δημιουργία ενός νέου καρτεσιανού communicator `new_comm` με `ndim` διαστάσεις
  - Αν `qperiodic[i]=1`, τότε στη διάσταση i το πλέγμα είναι περιοδικό

```
MPI_Comm cart_comm;
int ndim = 2;
int dims[2]= {4,4};
int qperiodic[2]= {0, 0};
int qreorder = 0;
MPI_Cart_create(MPI_COMM_WORLD,
      ndim, dims, qperiodic,
      qreorder, &cart_comm);
```

Καρτεσιανή τοπολογία στο MPI

- `MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])`
  - Εύρεση των καρτεσιανών συντεταγμένων `coords` της διεργασίας με βαθμό `rank` στο καρτεσιανό πλέγμα `comm`

- `MPI_Cart_rank(MPI_Comm comm, const int coords[], int * rank)`
  - Εύρεση του βαθμού `rank` μιας διεργασίας με συντεταγμένες `coords` στο καρτεσιανό πλέγμα `comm`

- `MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int * source, int * dest)`
  - Εύρεση της διεργασίας `dest` που απέχει απόσταση `displ` στην κατεύθυνση `direction` από τη διεργασία `source` στο καρτεσιανό πλέγμα `comm`

Μεταγλώττιση προγράμματος MPI στην ουρά parlab

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N compile_test

## Output and error files
#PBS -o testjob.out
#PBS -e testjob.err

## Limit memory, runtime etc.
#PBS -l walltime=01:00:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=1:ppn=1

module load openmpi/1.8.3
mpicc -O3 -Wall test.c -o test
```

Εκτέλεση προγράμματος MPI στην ουρά parlab

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N testjob

## Output and error files
#PBS -o testjob.out
#PBS -e testjob.err

## Limit memory, runtime etc.
#PBS -l walltime=01:00:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=8:ppn=8

module load openmpi/1.8.3
mpirun -np 64 –map-by node /home/users/nikela/test
```

MPI-2

- Το MPI-2 (MPI-2.0 το 1997, MPI-2.2 το 2009!) εισήγαγε πολλά νέα στοιχεία στο MPI:
  - Παράλληλη Είσοδος/Έξοδος (Parallel I/O)
    - Κλιμακωσιμότητα εφαρμογών που απαιτούν I/O
    - Περιορισμοί του υλικού
    - Checkpointing
  - Δυναμική Διαχείριση Διεργασιών (Dynamic Process Management)
    - Ικανοποίηση αναγκών συγκεκριμένων κλάσεων εφαρμογών
  - One-sided επικοινωνία (RMA – Remote Memory Access)
    - Διαφορετικό προγραμματιστικό μοντέλο
    - Πλέον συμβατό με το σύγχρονο hardware
    - Αναθεωρήθηκε στο MPI-3
  - MPI+Threads
    - Για υβριδικό προγραμματισμό
      - OpenMP/Pthreads εντός του κόμβου
      - MPI εκτός του κόμβου

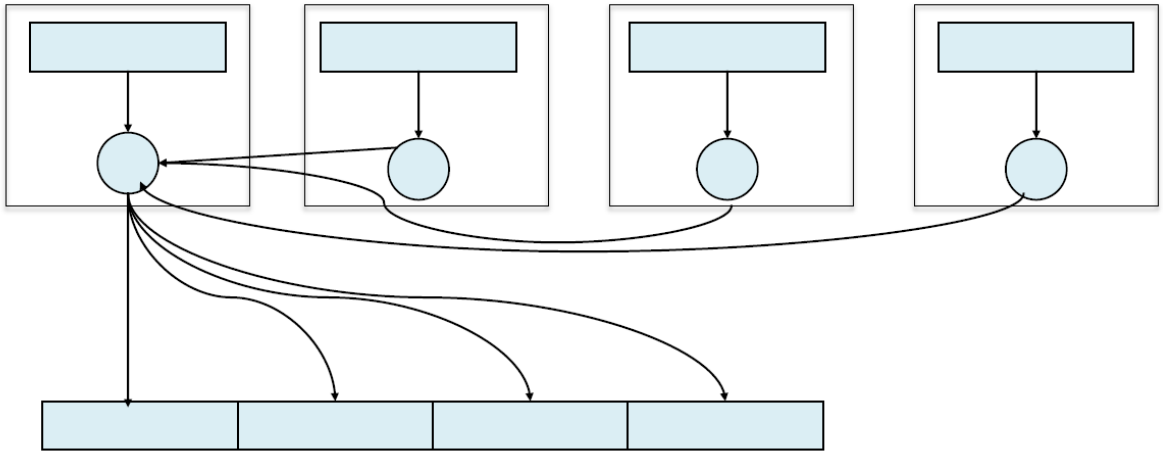
MPI I/O

- Οι παράλληλες εφαρμογές εκτελούν Ε/Ε για δύο βασικούς λόγους:
  - Ανάγνωση των αρχικών συνθηκών/δεδομένων
  - Εγγραφή αριθμητικών αποτελεσμάτων των προσομοιώσεων

- *Σειριακή ανάγνωση:* Μία διεργασία διαβάζει τα αρχικά δεδομένα και τα μοιράζει στις υπόλοιπες διεργασίες
- *Σειριακή εγγραφή:* Μία διεργασία συγκεντρώνει τα αποτελέσματα από τις διεργασίες και τα αποθηκεύει σε αρχείο

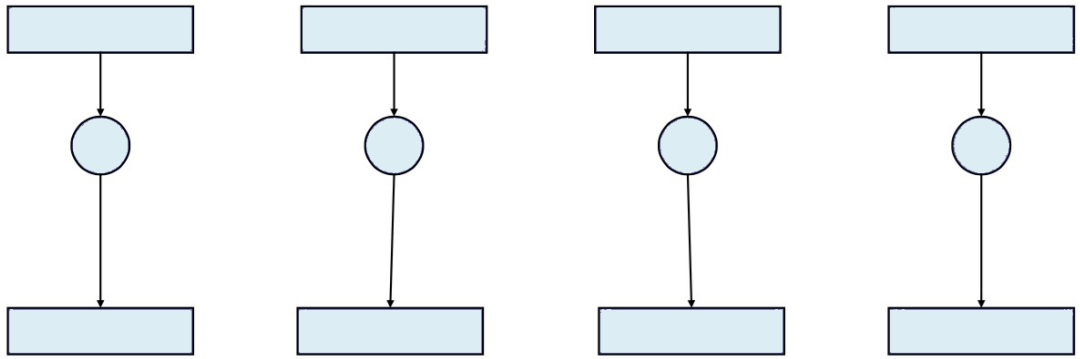
- ***Στον πραγματικό κόσμο, σπάνια τα δεδομένα χωράνε στη μνήμη μίας διεργασίας!***
  - *Μία αργή λύση:* Μία διεργασία διαβάζει τμηματικά τα αρχικά δεδομένα και τα στέλνει στις άλλες διεργασίες

Σειριακή Ε/Ε



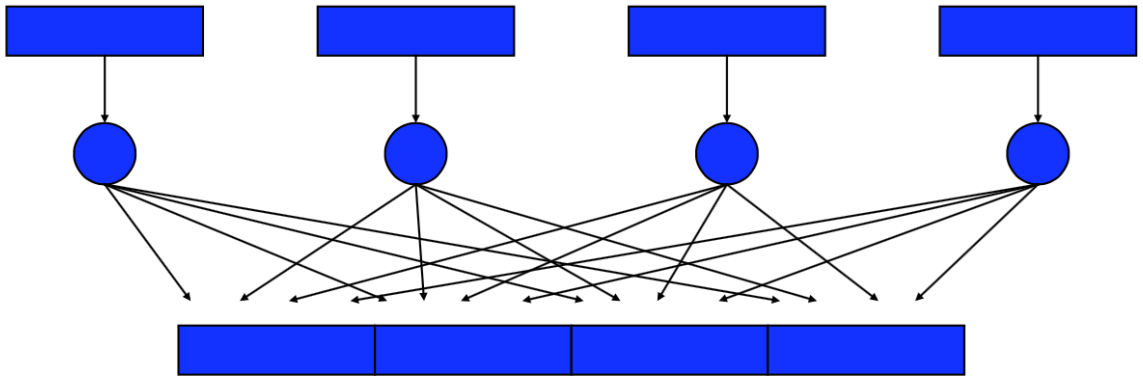
- Μία διεργασία συγκεντρώνει δεδομένα από τις άλλες διεργασίες και γράφει σε αρχείο
  - Μη παράλληλη Ε/Ε
  - Προκαλεί σειριοποίηση ή bottleneck

Ανεξάρτητη Παράλληλη Ε/Ε



- Κάθε διεργασία γράφει σε ξεχωριστό αρχείο
  - Παράλληλη Ε/Ε
  - Απαιτεί διαχείριση πολλών μικρών αρχείων

Παράλληλη Ε/Ε



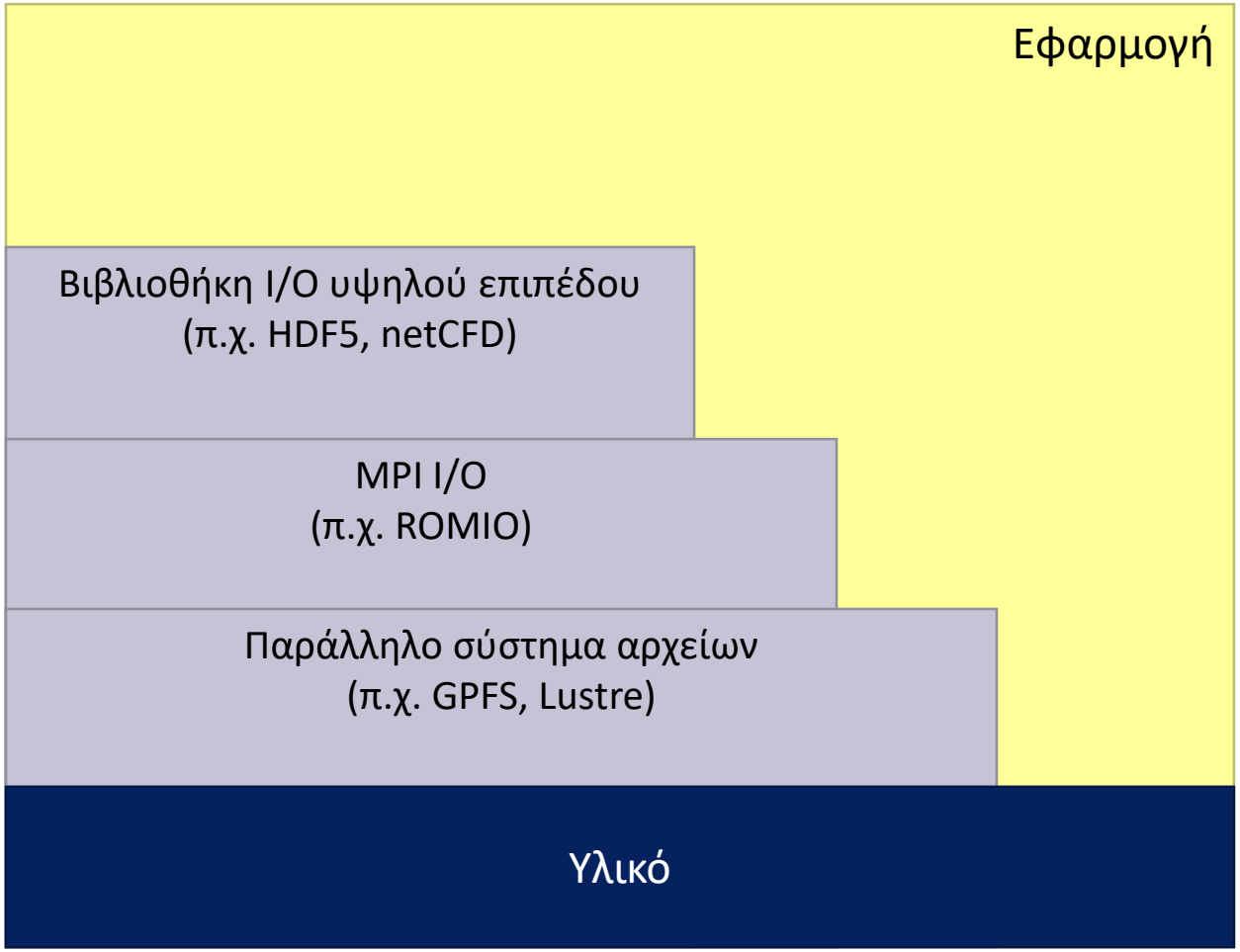
- Όλες οι διεργασίες γράφουν παράλληλα και «συνεργατικά» στο ίδιο αρχείο
  - Παράλληλη Ε/Ε

Παράλληλη Ε/Ε στο MPI

- Απαραίτητοι μηχανισμοί για παράλληλη Ε/Ε:
  - Collective operations
  - Τύποι δεδομένων για μη συνεχόμενα δεδομένα στη μνήμη/σε αρχείο
    - Παρόμοιοι με τους μηχανισμούς του MPI για επικοινωνία!

- Από την πλευρά του συστήματος, απαιτείται παράλληλο σύστημα αρχείων και υλικό που επιτρέπει παράλληλες προσβάσεις

HW/SW stack για MPI I/O



MPI I/O: Βασικές λειτουργίες

- `MPI_File_open` για παράλληλο άνοιγμα αρχείου με τα σχετικά modes
- Παράλληλη ανάγνωση αρχείου με
  - `MPI_File_seek` και `MPI_File_read` ή
  - `MPI_File_read_at`
- Παράλληλη εγγραφή αρχείου με
  - `MPI_File_set_view` και `MPI_File_write` ή
  - `MPI_File_write_at`
- Παράλληλο κλείσιμο αρχείου με `MPI_File_close`

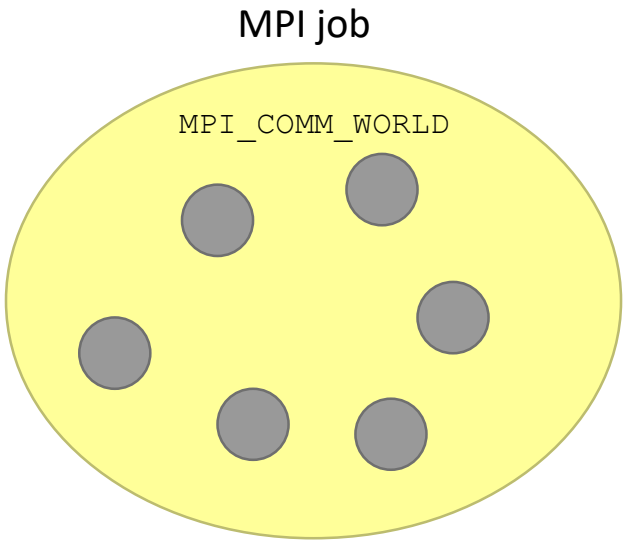
- ...και πολλές συναρτήσεις για collective I/O

Δυναμική διαχείριση διεργασιών

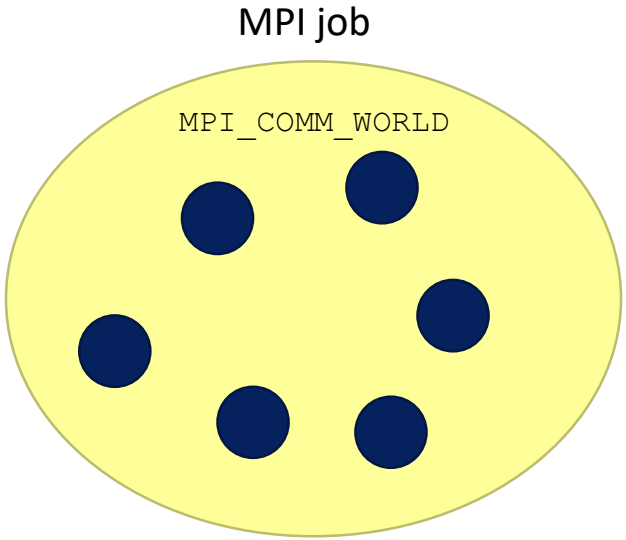
- Η δυναμική διαχείριση διεργασιών επιτρέπει:
  - Την προσθήκη διεργασιών σε κάποια δουλειά που ήδη τρέχει
    - Σε εφαρμογές που χρειάζονται περισσότερες διεργασίες
      - Π.χ. σε έναν αλγόριθμο branch-and-bound
    - Σε παράλληλο προγραμματισμό τύπου master-slave
    - Όταν νέοι πόροι γίνονται διαθέσιμοι
  - Τη συνένωση εφαρμογών που τρέχουν ξεχωριστά
    - Client-server, Peer-to-peer κ.ά.
  - Τη διαχείριση σφαλμάτων

- Το MPI αναλαμβάνει την επικοινωνία μεταξύ ομάδων διεργασιών από διαφορετικές εργασίες
  - Δεν αναλαμβάνει τη δημιουργία νέων διεργασιών ή την εκκίνηση νέων εργασιών
    - Αυτό αποτελεί ευθύνη του resource manager (π.χ. Torque)

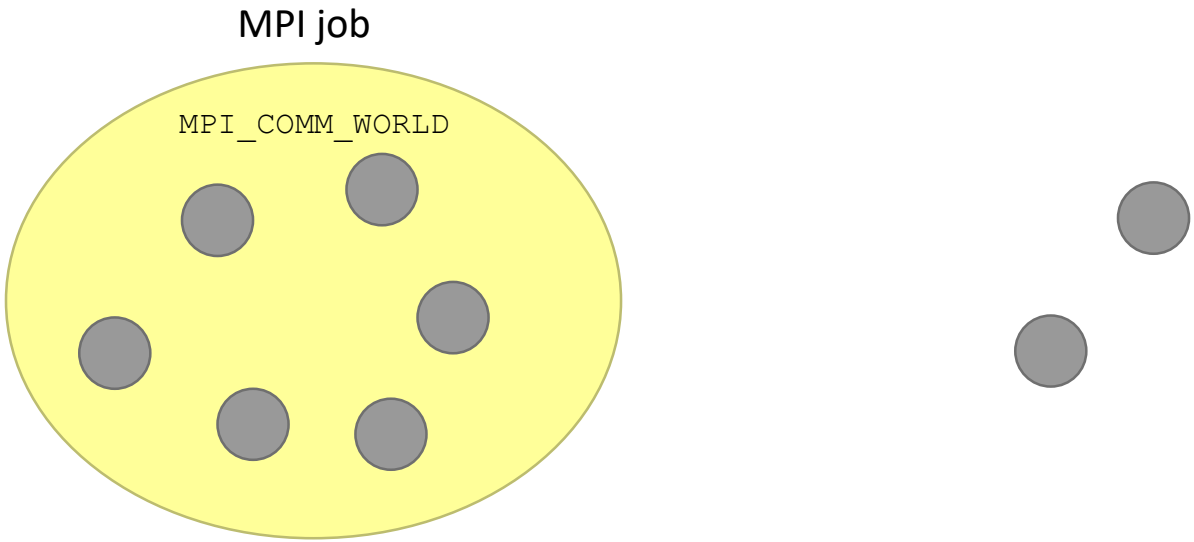
Δυναμική δημιουργία διεργασιών στο MPI



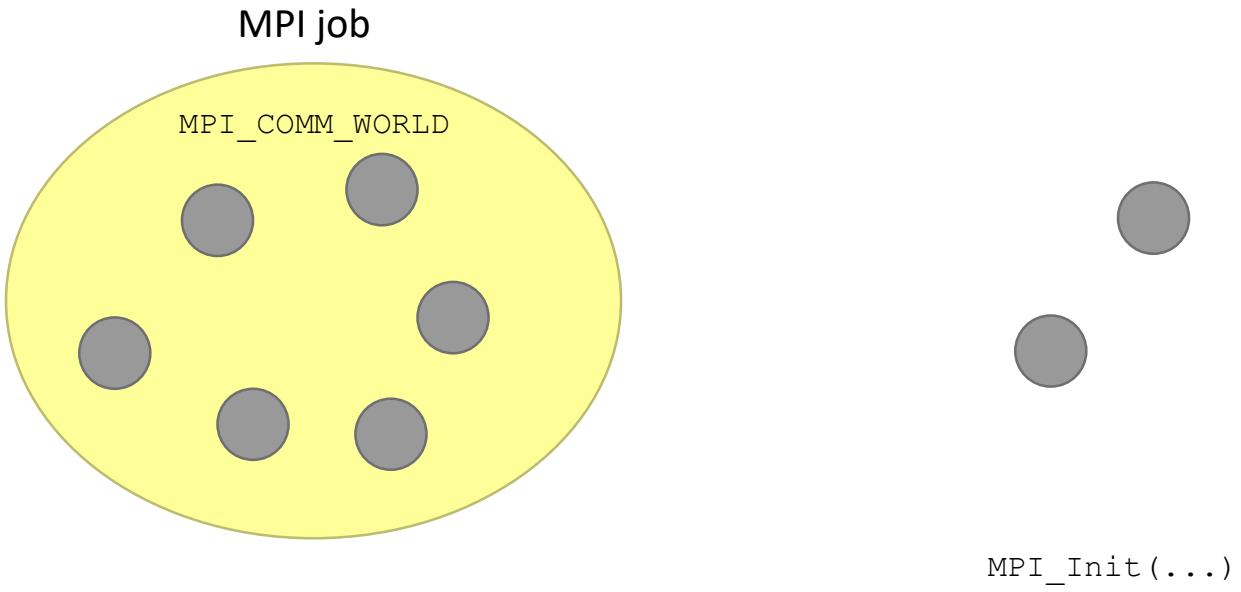
Δυναμική δημιουργία διεργασιών στο MPI



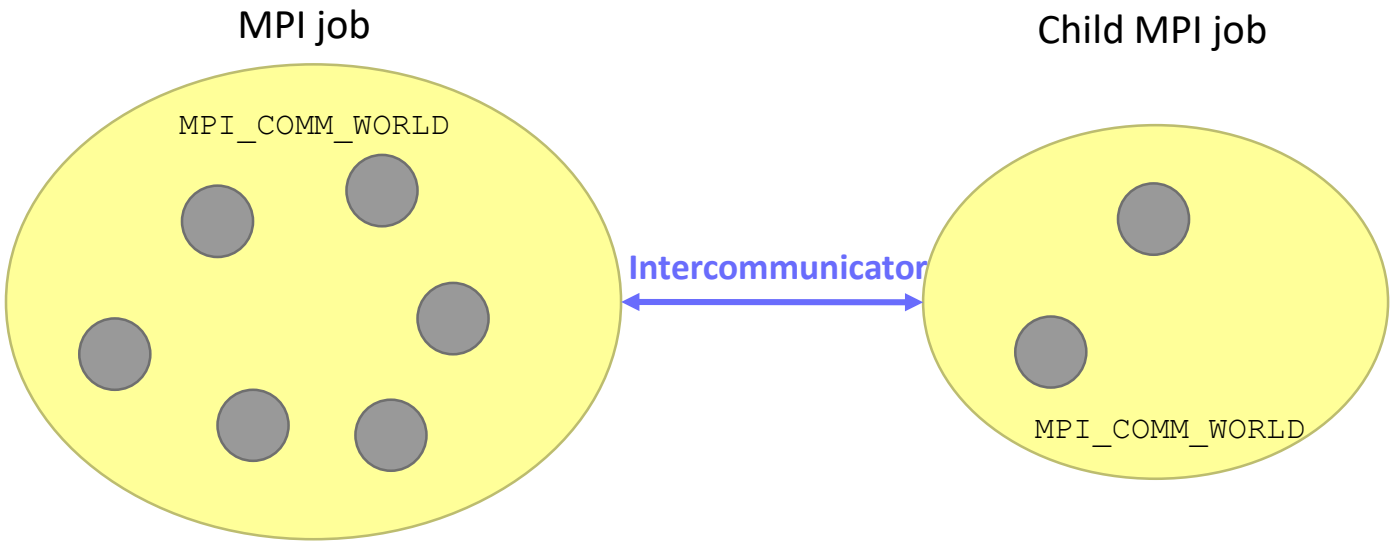
Δυναμική δημιουργία διεργασιών στο MPI



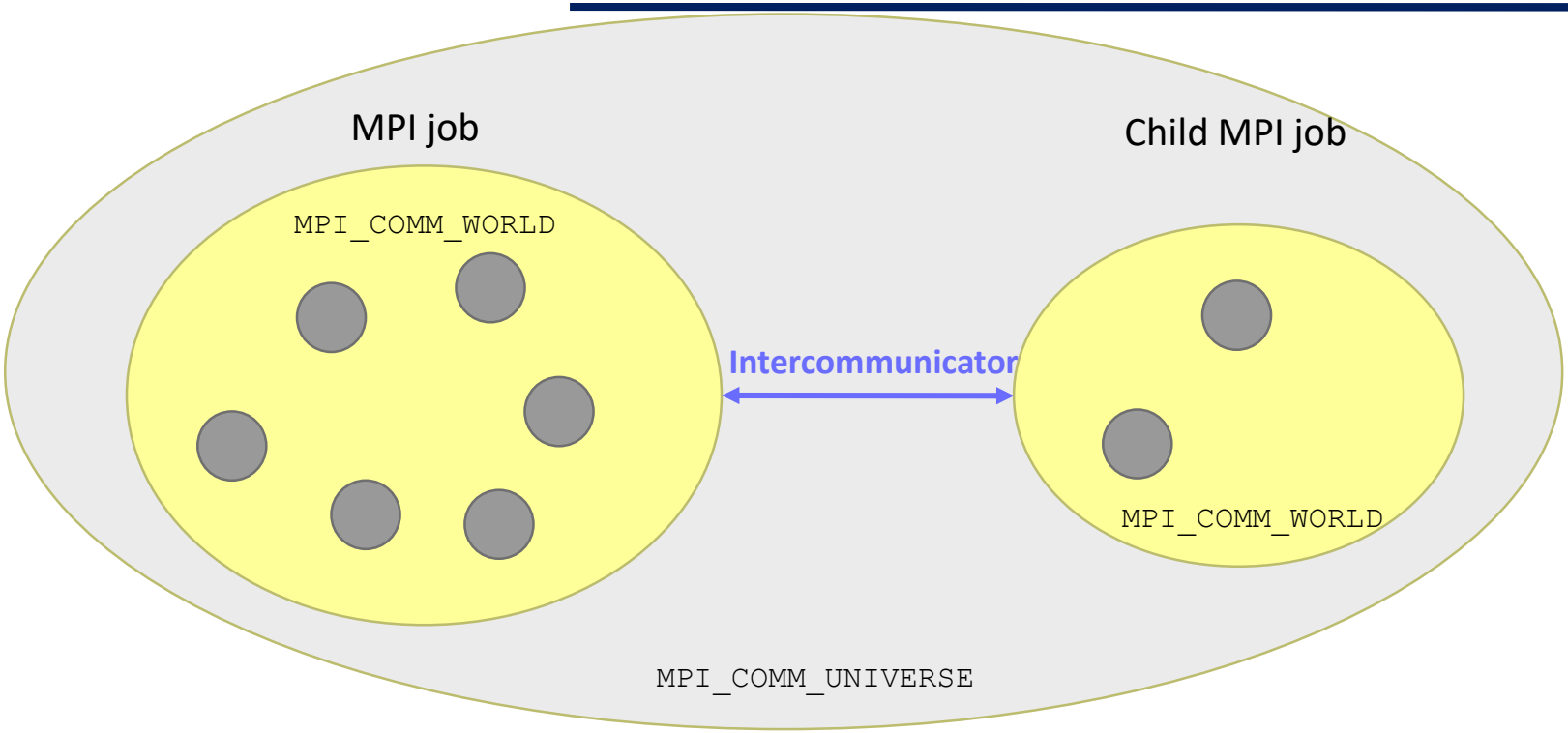
Δυναμική δημιουργία διεργασιών στο MPI



Δυναμική δημιουργία διεργασιών στο MPI



Δυναμική δημιουργία διεργασιών στο MPI



Επικοινωνία και δυναμική διαχείριση διεργασιών

- Οι διεργασίες πολλαπλών συνδεδεμένων εργασιών MPI επικοινωνούν μέσω συναρτήσεων του MPI πάνω από τον intercommunicator
  - Ο intercommunicator έχει μέγεθος `MPI_UNIVERSE_SIZE`
  - Οι διεργασίες παίρνουν αναγνωριστικά από `0` ως `MPI_UNIVERSE_SIZE-1`

- Εκτός από δημιουργία νέας ομάδας διεργασιών με `MPI_Comm_spawn`, παρέχονται συναρτήσεις τύπου server-client για την επικοινωνία διαφορετικών εργασιών
  - Server-side: `MPI_Open_port`, `MPI_Comm_accept`, `MPI_Close_port`
  - Client-side: `MPI_Comm_connect`

Δίπλευρη (point-to-point) vs Μονόπλευρη (one-sided) επικοινωνία

- *Point-to-point* επικοινωνία:
  - Απαιτείται διμερής γνώση!
    - Οι δύο διεργασίες πρέπει να γνωρίζουν ποιες θέσεις μνήμης τροποποιούνται
    - Οι δύο διεργασίες πρέπει να γνωρίζουν πότε θα συμβεί η τροποποίηση
    - Οι δύο διεργασίες πρέπει να «συμφωνήσουν» ότι θα επικοινωνήσουν
      - **Send** ΚΑΙ **Recv**

- *One-sided* επικοινωνία:
  - Η γνώση είναι μονομερής!
    - Μόνο μία διεργασία ορίζει από πού προέρχονται και πού καταλήγουν τα δεδομένα
    - Αυτή η διεργασία κάνει **πρόσβαση σε απομακρυσμένο χώρο διευθύνσεων**
      - **RMA –** Remote Memory Access
      - **Put** Ή **Get**
  - Διαισθητικά πιο κοντά σε μοντέλο κοινού χώρου διευθύνσεων

Δίπλευρη (point-to-point) vs Μονόπλευρη (one-sided) επικοινωνία

- *Point-to-point* επικοινωνία:
  - Απαιτείται διμερής γνώση!

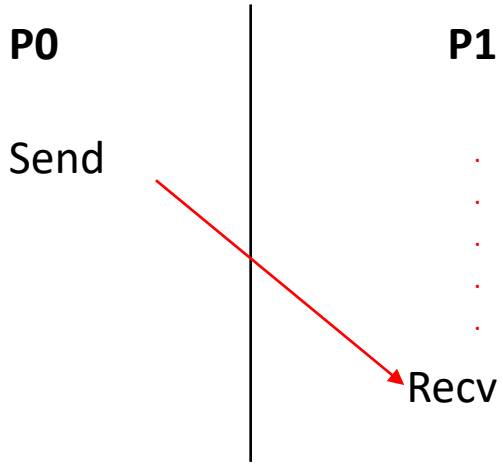
P0	P1
store	
send	recv
	load

- *One-sided* επικοινωνία:
  - Η γνώση είναι μονομερής!

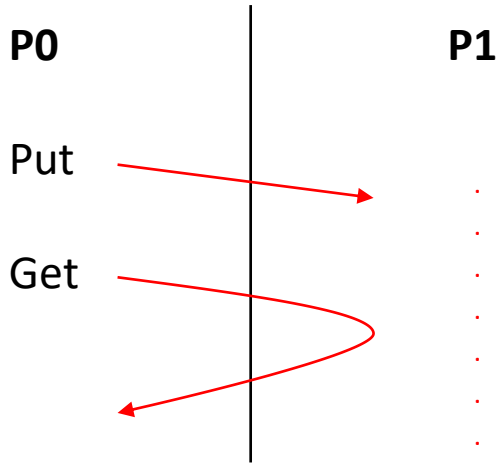
P0	P1
fence	fence
put	
fence	fence
	load

P0	P1
store	
fence	fence
	get
	fence

Δίπλευρη (point-to-point) vs Μονόπλευρη (one-sided) επικοινωνία

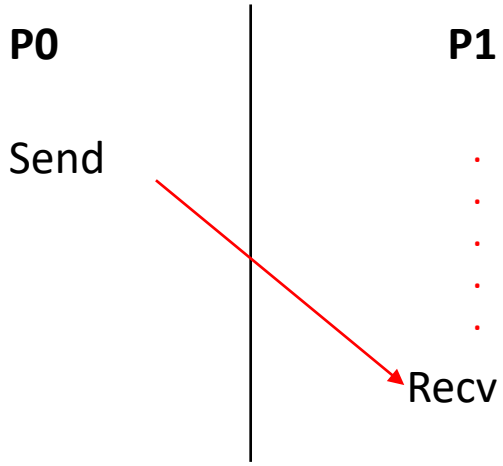


- Στη δίπλευρη επικοινωνία, καθυστέρηση στην P1 επηρεάζει την P0

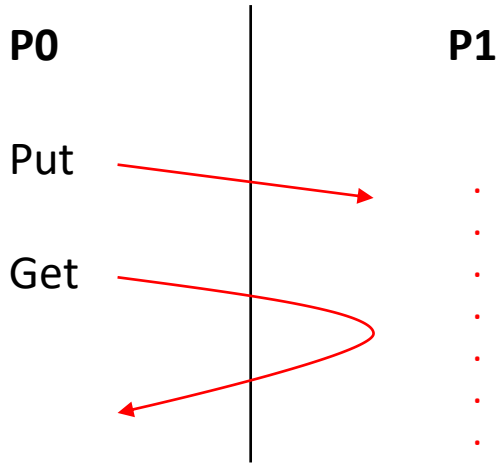


- Στη μονόπλευρη επικοινωνία, οποιαδήποτε καθυστέρηση στην P1 δεν επηρεάζει την P0

Δίπλευρη (point-to-point) vs Μονόπλευρη (one-sided) επικοινωνία

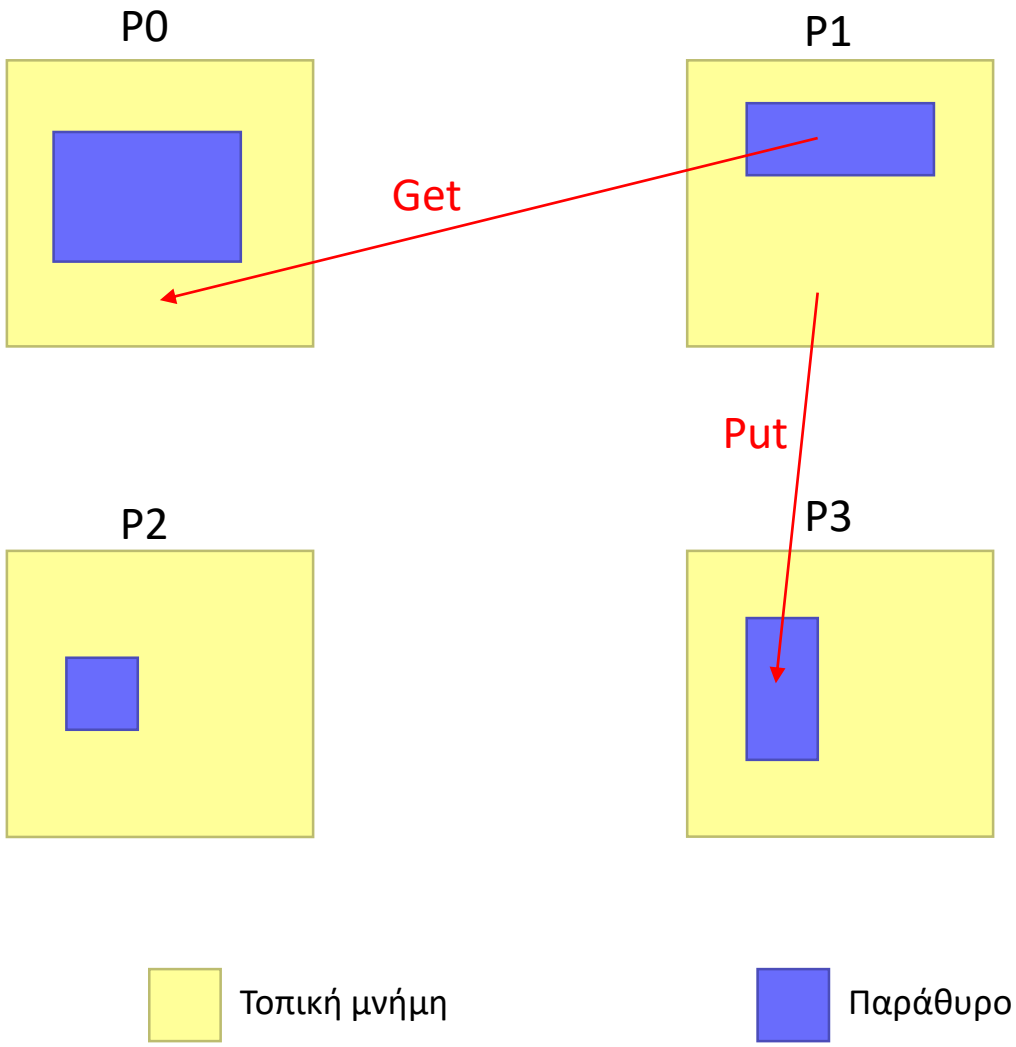


- Στη δίπλευρη επικοινωνία, καθυστέρηση στην P1 επηρεάζει την P0



- Στη μονόπλευρη επικοινωνία, οποιαδήποτε καθυστέρηση στην P1 δεν επηρεάζει την P0

Πρόσβαση σε απομακρυσμένη μνήμη στο MPI



One-sided communication: MPI_Win

- Για να είναι δυνατή η one-sided επικοινωνία, κάθε διεργασία πρέπει να «ανοίξει ένα παράθυρο» - να δηλώσει ένα μέρος της μνήμης της στο οποίο επιτρέπει απομακρυσμένες προσβάσεις

```
MPI_Win_create (
    void * base,
    MPI_Aint size,
    int disp_unit,
    MPI_Info info,
    MPI_Comm comm,
    MPI_Win * win)
```

- Η διεργασία ορίζει ένα «παράθυρο» `win` στη μνήμη της που χρησιμοποιείται από τις άλλες διεργασίες στον communicator `comm` για one-sided επικοινωνία
  - Καλείται συλλογικά από τις διεργασίες του communicator
  - Δημιουργείται το αντικείμενο `win`

```
MPI_Win_free(MPI_Win * win)
```

MPI_Put: Εγγραφή σε απομακρυσμένη μνήμη

```
MPI_Put(
    const void * sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    int dest,
    MPI_Aint dest_disp,
    int recvcount,
    MPI_Datatype recvtype,
    MPI_Win win)
```

- Η διεργασία μεταφέρει («βάζει») δεδομένα από την τοπική της μνήμη στην απομακρυσμένη μνήμη
  - Η διεργασία μεταφέρει από τον buffer `sendbuf` δεδομένα πλήθους `sendcount` και τύπου `sendtype` στη διεργασία με rank `dest`

MPI_Get: Ανάγνωση από απομακρυσμένη μνήμη

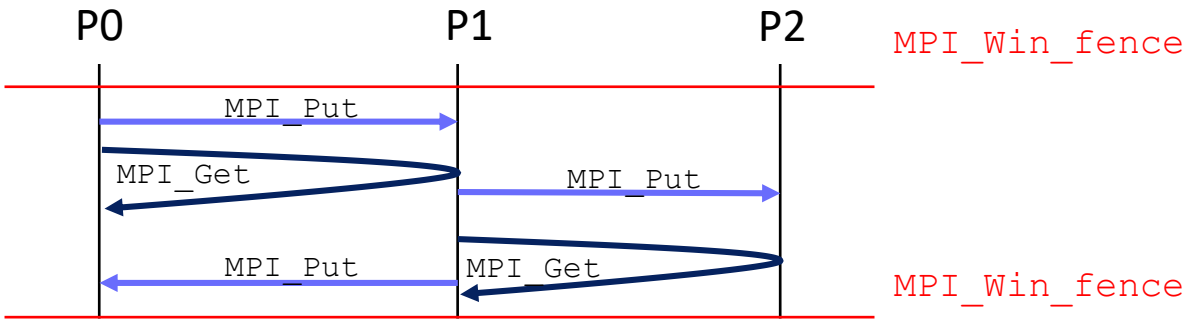
```
MPI_Get(
    const void * recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    int source,
    MPI_Aint source_disp,
    int sendcount,
    MPI_Datatype sendtype,
    MPI_Win win)
```

- Η διεργασία μεταφέρει («παίρνει») δεδομένα από την απομακρυσμένη μνήμη στην τοπική της μνήμη
  - Η διεργασία μεταφέρει στον buffer `recvbuf` δεδομένα πλήθους `recvcount` και τύπου `recvtype` από τη διεργασία με rank `source`

Συλλογικός συγχρονισμός στην one-sided επικοινωνία

```
MPI_Win_fence(int assert, MPI_Win win)
```

- Ξεκινά και ολοκληρώνει «εποχές» (epochs) συλλογικής επικοινωνίας
  - Συλλογικός συγχρονισμός
- Όλες οι διεργασίες με πρόσβαση στο αντικείμενο win καλούν την MPI_Win_fence για να ξεκινήσουν μία εποχή
- Οι διεργασίες καλούν τις MPI_Put/MPI_Get για να γράψουν/διαβάσουν σε/από απομακρυσμένη μνήμη
- Όλες οι διεργασίες καλούν ξανά την MPI_Win_fence για να ολοκληρωθεί η εποχή
  - Ολοκληρώνονται όλες οι εντολές απομακρυσμένης εγγραφής/ανάγνωσης της εποχής
- Λέγεται και «ενεργός συγχρονισμός»
  - Οι διεργασίες γνωρίζουν ότι ξεκινά και τελειώνει μία εποχή προσβάσεων

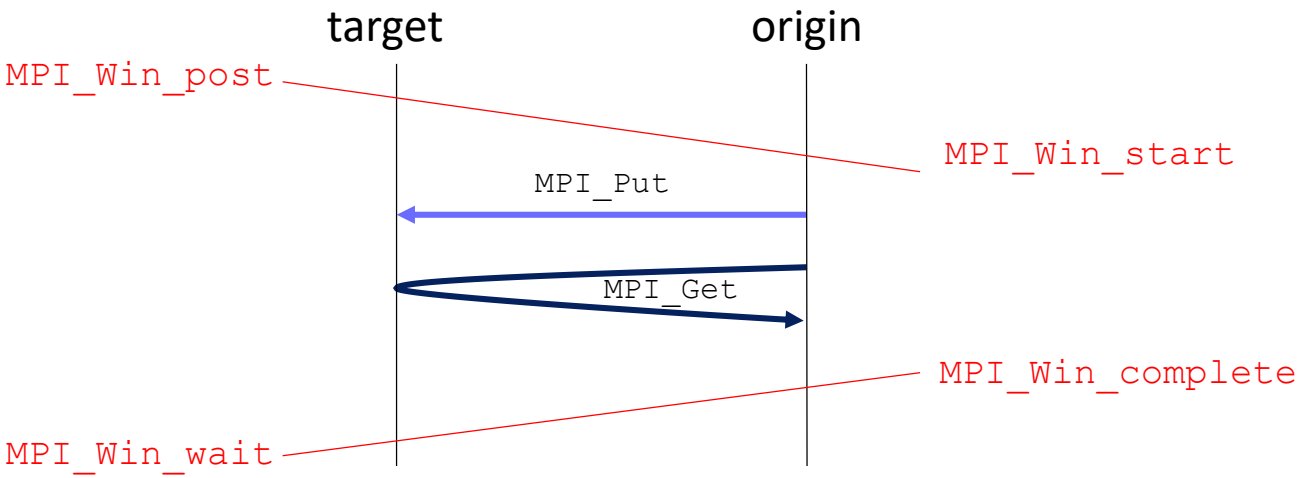


Ενεργός συγχρονισμός στην one-sided επικοινωνία

`MPI_Win_post/start(MPI_Group group, int assert, MPI_Win win)`

`MPI_Win_complete/wait(MPI_Win win)`

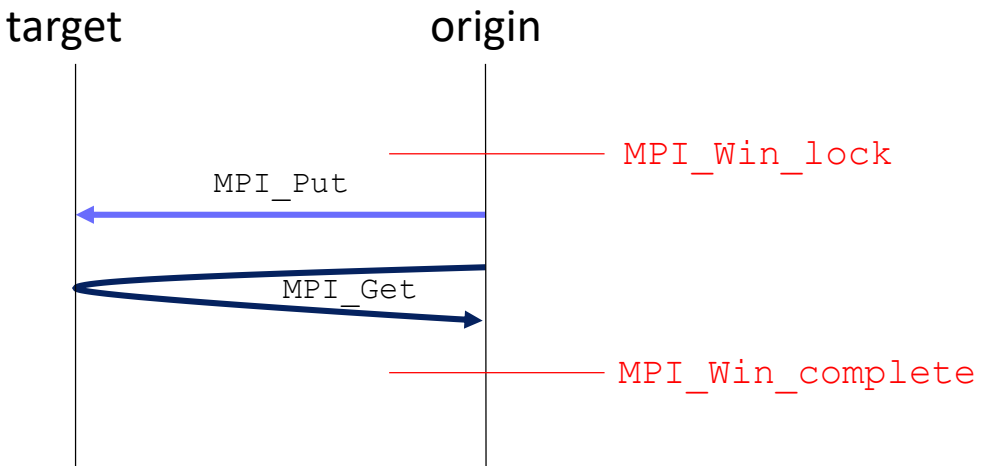
- Μία διεργασία `target` ξεκινά μία εποχή με `MPI_Win_post` και ολοκληρώνει μία εποχή με `MPI_Win_wait` κατά την οποία εκθέτει το αντικείμενο `win` σε όλες τις διεργασίες του group `group`
- Στη διάρκεια αυτής της εποχής, μία άλλη διεργασία `origin` μπορεί να εκκινήσει μία εποχή προσβάσεων στο παράθυρο της διεργασίας `target` με κλήση `MPI_Win_start` και να την ολοκληρώσει με κλήση `MPI_Win_complete`



Παθητικός συγχρονισμός στην one-sided επικοινωνία

```
MPI_Win_lock(int locktype, int target, int assert, MPI_Win win)
MPI_Win_unlock(int target, MPI_Win win)
```

- Μία διεργασία `origin` ξεκινά μία εποχή κλειδώνοντας το αντικείμενο `win` της διεργασίας `target` με `MPI_Win_lock` και ολοκληρώνει την εποχή με `MPI_Win_unlock`
- Η διεργασία `target` δεν μετέχει στο συγχρονισμό



Πλεονεκτήματα της μονόπλευρης επικοινωνίας

- Επιτρέπει πολλαπλές μεταφορές δεδομένων με μία μόνο κλήση για συγχρονισμό
- Δεν απαιτεί tag-matching
  - Tag matching: ταίριασμα ενός μηνύματος που παρελήφθη από μία διεργασία με την αντίστοιχη κλήση Recv
- Βοηθά στην έκφραση ασύμμετρων/ακανόνιστων σχημάτων επικοινωνίας
  - Κατανεμημένες δομές δεδομένων κ.ά.
  - Αν δεν είναι γνωστό ποιες διεργασίες θα επικοινωνήσουν με ποιες, αλλά είναι γνωστές οι θέσεις μνήμης όπου θα μεταφερθούν δεδομένα, τότε βοηθά την έκφραση της επικοινωνίας
- Μπορεί να είναι αρκετά ταχύτερη **αν υποστηρίζεται από το υλικό**
  - Από το δίκτυο διασύνδεσης

Μειονεκτήματα της μονόπλευρης επικοινωνίας

- Απαιτείται συγχρονισμός, που δεν κλιμακώνει πάντα αποδοτικά
- Οι προγραμματιστές θεωρούν τις one-sided συναρτήσεις του MPI περίπλοκες και μη αποδοτικές
  - Τροποποιήσεις στο MPI-3
- Αν δεν υποστηρίζεται από το υλικό, δεν επιτυγχάνει υψηλή επίδοση
  - Όταν προτάθηκε το MPI-2 δεν υπήρχε υποστήριξη από το υλικό

MPI-3

- Το MPI-3 προτάθηκε το 2012 (MPI-3.1 τον Ιούνιο του 2015) με στόχο να λύσει νέα προβλήματα και να ανταποκριθεί στις αλλαγές του υλικού
  - Non-blocking collectives
    - Η συλλογική επικοινωνία καταναλώνει σημαντικό μέρος του χρόνου εκτέλεσης των εφαρμογών σε συστήματα μεγάλης κλίμακας
    - Η ασύγχρονη συλλογική επικοινωνία βοηθά στο να κρύψει/επικαλύψει μέρος του χρόνου αυτού
  - Αραιή συλλογική επικοινωνία (neighborhood collectives)
    - Καλύτερη κλιμάκωση – ευκολότερος προγραμματισμός για εφαρμογές που επικοινωνούν συλλογικά με γείτονες (Jacobi κ.ά.)
  - Σημαντικές επεκτάσεις στην one-sided επικοινωνία
    - Για καλύτερη επίδοση σε υβριδικά συστήματα
    - Για διαχείριση παράλληλων δομών δεδομένων
    - Πιο συμβατή με την υποστήριξη από το υλικό (υπάρχον και μελλοντικό)
  - Υποστήριξη για εργαλεία επίδοσης
  - Καλύτερη διαχείριση νημάτων

Non-blocking collectives στο MPI-3

- Η συλλογική επικοινωνία κλιμακώνει λογαριθμικά στην καλύτερη περίπτωση
- Η ασύγχρονη συλλογική επικοινωνία επιτρέπει επικαλύψεις με υπολογισμούς
- Τα non-blocking collectives προέρχονται από την εμπειρία εφαρμογών του πραγματικού κόσμου
  - `MPI_Ialltoall` – Fast Fourier Transforms
  - `MPI_Iallreduce` – Krylov methods
  - `MPI_Iscatter, MPI_Igather` – stencils
  - ... και άλλα, όπως `MPI_Ibcast, MPI_Ibarrier`

- Τα νεότερα δίκτυα διασύνδεσης υλοποιούν συναρτήσεις συλλογικής επικοινωνίας απευθείας στο hardware
  - Με ασύγχρονη συλλογική επικοινωνία, ο επεξεργαστής χρησιμοποιείται ελάχιστα

Neighborhood collectives στο MPI-3

- Σε πολλές εφαρμογές, οι διεργασίες επικοινωνούν επαναληπτικά με συγκεκριμένους γείτονες
- Τα neighborhood collectives προσφέρουν προγραμματιστική ευκολία
  - Ο προγραμματιστής ορίζει γράφους γειτόνων
    - `MPI_Dist_graph_neighbors`
  - Οι πολλές κλήσεις της point-to-point επικοινωνίας αντικαθίστανται από μία γραμμή
    - `MPI_Neighbor_allgather`, `MPI_Neighbor_allgatherv`, `MPI_Neighbor_alltoall`, `MPI_Neighbor_alltoallv` και non-blocking εκδοχές
- Η υλοποίηση του MPI αναλαμβάνει τη βέλτιστη υλοποίηση του σχήματος επικοινωνίας
  - Με επίγνωση και της αρχιτεκτονικής και των δυνατοτήτων του υλικού

Επεκτάσεις στην one-sided επικοινωνία στο MPI-3

- Η one-sided επικοινωνία του MPI-2 βοηθά στη διατήρηση κατανεμημένων δομών δεδομένων

- Το MPI-3 προσθέτει λειτουργίες που σχετίζονται με το συγχρονισμό
  - Non-blocking locks (κάποια ήδη από το MPI-2)
    - `MPI_Win_lock/unlock, MPI_Win_lock_all/unlock_all` κ.ά.
  - Atomic operations
    - `MPI_Fetch_and_op, MPI_Compare_and_swap` κ.ά.

- Πολλές νέες λειτουργίες υποστηρίζονται πλέον απευθείας από το δίκτυο διασύνδεσης

Ερωτήσεις;