9 & 16/5/2023

# Προγραμματισμός με μεταβίβαση μηνυμάτων

Λ8

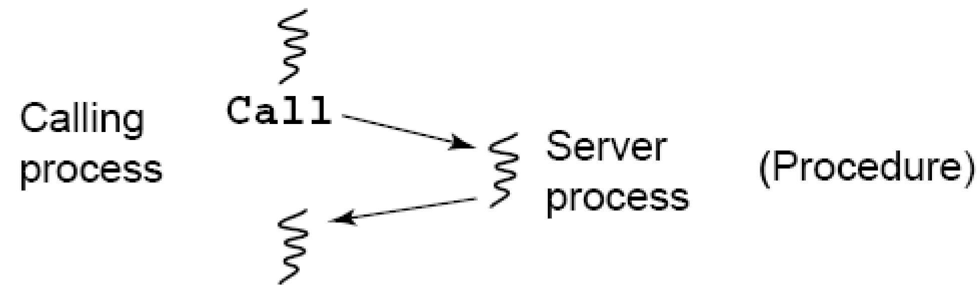
Συστήματα & Λογισμικό Υψηλών Επιδόσεων

## «Κατηγορίες» προγραμματισμού κατανεμημένης μνήμης

- Sockets
  - Απλά το αναφέρουμε…
- Message passing (μεταβίβαση μηνυμάτων)
  - με αυτό θα ασχοληθούμε
  - πιο «χαμηλό» επίπεδο από τα παρακάτω (αλλά και πιο βολικό για τις περισσότερες εφαρμογές)
  - στηρίζεται σε 2 συναρτήσεις: μία για αποστολή μηνύματος και μία για λήψη μηνύματος (μεταξύ διεργασιών)
- Remote procedure calls (RPC)
  - όχι μηνύματα – εκτέλεση συναρτήσεων σε απομακρυσμένο κόμβο
  - Π.χ. SUN RPC (rpcgen), Java RMI
- Rendezvous
  - «σύγχρονο» PRC

# RPC

- Στον “server” πρέπει να έχουν δηλωθεί όλες οι συναρτήσεις που μπορούν να εκτελεστούν από τον “client”.
- O client
  1. καλεί την απομακρυσμένη ρουτίνα (όπου μαζί στέλνει και τα δεδομένα – παραμέτρους της ρουτίνας)
  2. μπλοκάρει περιμένοντας τα αποτελέσματα
  3. παραλαμβάνει τα αποτελέσματα και συνεχίζει
- O προγραμματιστής *δεν εμπλέκεται καθόλου στις αποστολές των δεδομένων* – απλά καλεί την απομακρυσμένη ρουτίνα σαν να ήταν μια τοπική συνάρτηση.



# Rendezvous

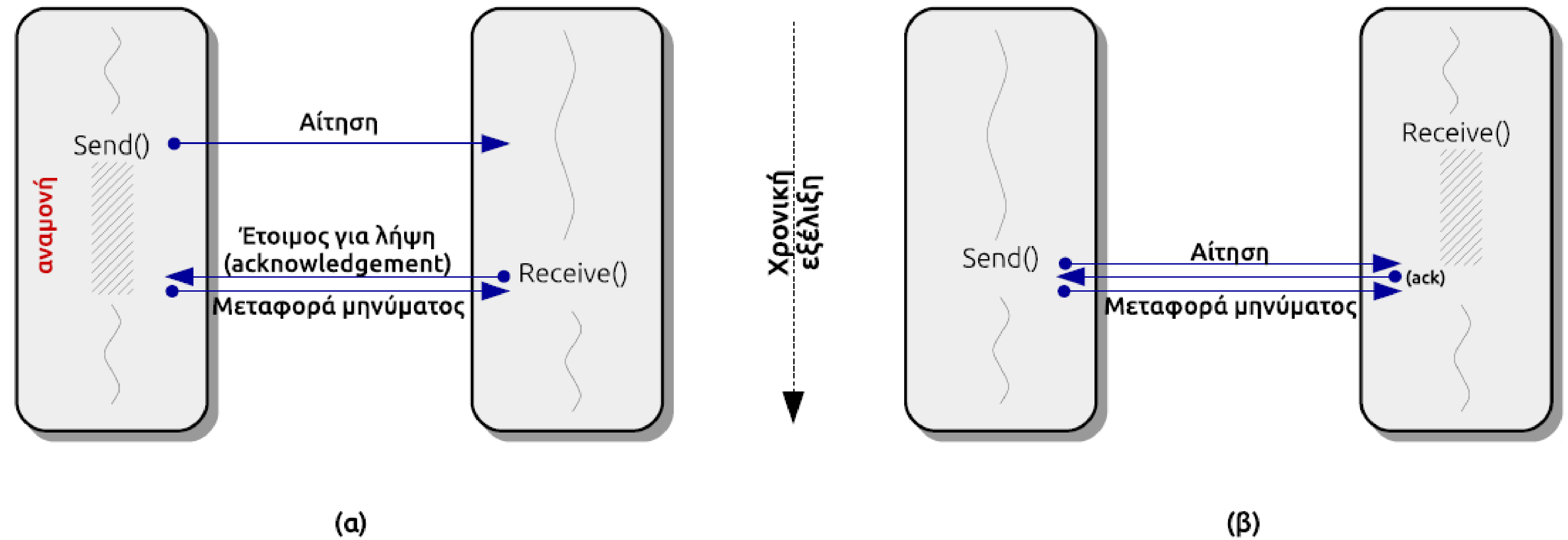
- Ο PRC server φτιάχνει ένα νέο thread για να εξυπηρετήσει τις αιτήσεις που έρχονται για κλήση των συναρτήσεών του (ασύγχρονη εξυπηρέτηση)
  - αυτό απαιτεί π.χ. αμοιβαίο αποκλεισμό αν στον server τροποποιούνται κοινά δεδομένα
- Στο rendezvous, ο server «μπλοκάρει» μέχρι να του έρθει μία αίτηση
  - επομένως μπορεί να εξυπηρετεί μόνο έναν client τη φορά
  - και άρα δεν χρειάζεται αμοιβαίος αποκλεισμός πουθενά
  - οι κλήσεις μεταξύ client / server πρέπει να είναι προκαθορισμένες και με δεδομένη σειρά

# Μεταβίβαση μηνυμάτων: μόνο 2 συναρτήσεις είναι αρκετές …

- Αποστολή μηνύματος – send()
- Παραλαβή μηνύματος – receive()
- Θέματα που μπαίνουν:
  - Ταχύτητα επικοινωνιών
    - Εξαρτάται από το δίκτυο και τα χαρακτηριστικά του
    - Εξαρτάται και από τη βιβλιοθήκη που υλοποιεί τις send() και receive()
      - buffering
      - copying
      - os traps / user-level access
  - «Στυλ» επικοινωνιών
    - Συγχρονισμένο ή τυπου rendezvous (το send() δεν ολοκληρώνεται αν ο παραλήπτης δεν έχει ξεκινήσει το receive())
    - Ασύγχρονο ή buffered
    - Blocking / non-blocking
    - Private / collective
- Πολλές επιλογές πλέον …
  - … η εξής μία: **MPI**

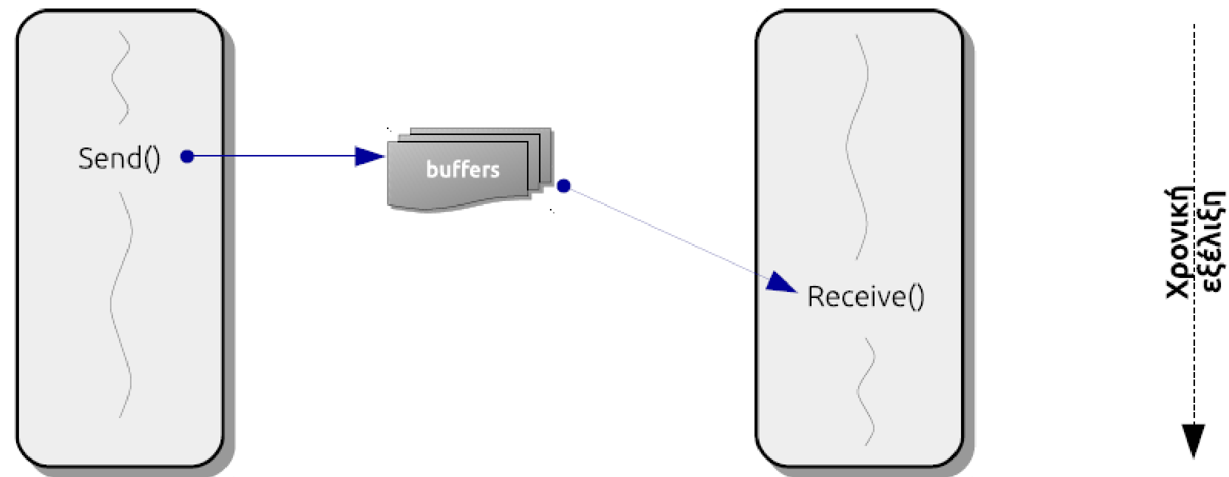
# Σύγχρονη επικοινωνία (τύπου rendezvous)

- Συγχρονισμένο ή τυπου rendezvous (το send() δεν ολοκληρώνεται αν ο παραλήπτης δεν έχει ξεκινήσει το receive())



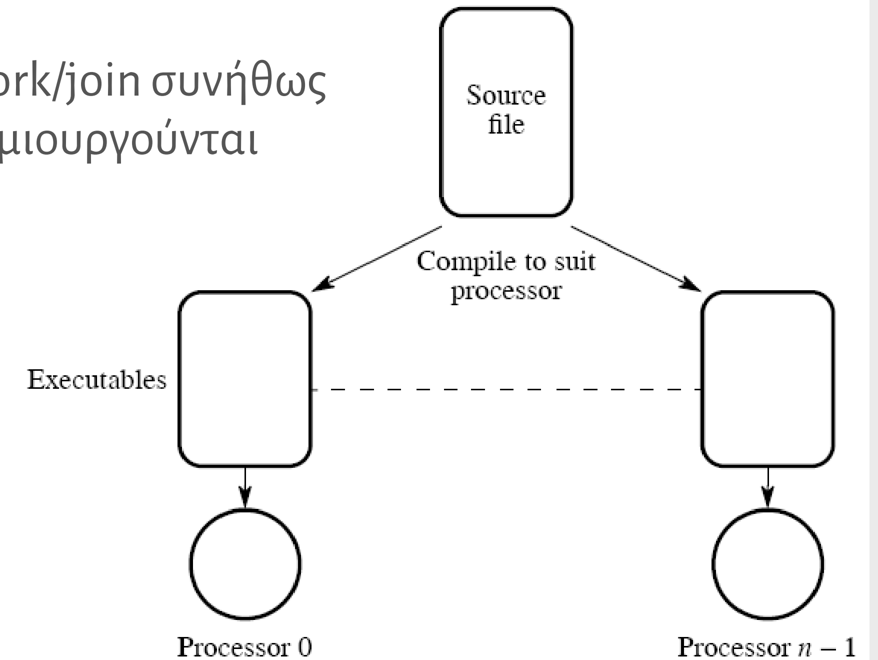
# «Κανονική» (ασύγχρονη) επικοινωνία

- Το send() ολοκληρώνεται άσχετα με το πότε θα γίνει το receive() και η διεργασία που στέλνει το μήνυμα συνεχίζει αμέσως.
- Η πιο συχνή στην πράξη
- Απαιτεί buffering
  - είτε αυτόματα από τη βιβλιοθήκη ("standard" mode στο MPI) είτε από τον προγραμματιστή ("buffered" mode στο MPI)



# MPI - Βασικές δομές

- Διεργασίες
  - … εξαρτάται από το σύστημα, όχι fork/join συνήθως
  - Η πιο κοινή περίπτωση είναι να δημιουργούνται *εξωτερικά*:
    - mpirun –np 10 a.out
    - «SPMD»




- Για διαφοροποίηση των διεργασιών:
  - `MPI_Comm_rank(MPI_COMM_WORLD, &myid);`
    - Ακολουθιακή αρίθμηση (0, 1, 2, …)
  - `MPI_Comm_size(MPI_COMM_WORLD, &nproc);`
    - Πλήθος διεργασιών συνολικά (το “-np” που δόθηκε παραπάνω)

# Βασικές λειτουργίες

- Αποστολή μηνυμάτων
  - `MPI_Send(buf, n, dtype, torank, tag, MPI_COMM_WORLD);`

  - buf: διεύθυνση του send buffer
  - n: το πλήθος των στοιχείων του buffer
  - dtype: ο τύπος των στοιχείων του buffer (MPI_CHAR/SHORT/INT/LONG/FLOAT/DOUBLE)
  - torank: id της διεργασίας που θα λάβει το μήνυμα
  - tag: ετικέτα (ότι θέλει βάζει ο προγραμματιστής)
- Λήψη μηνυμάτων
  - `MPI_Recv(buf, n, dtype, fromrank, tag, MPI_COMM_WORLD, status);`

  - buf: διεύθυνση του receive buffer
  - fromrank: id της διεργασίας που θα στείλει το μήνυμα
  - status: διεύθυνση buffer για πληροφορίες σε σχέση με το παραληφθέν μήνυμα
- Παραλαβή ΜΟΝΟ ΕΦΟΣΟΝ:
  - το μήνυμα όντως ήρθε από τη διεργασία fromrank
  - το μήνυμα είχε όντως την ετικέτα tag

# «Τυφλή» λήψη

- Πολλές φορές χρήσιμο να παραλάβουμε όποιο μήνυμα μας έρθει πρώτο, άσχετα ποιος το έστειλε και με τι ετικέτα:
  - `MPI_Recv(buf, n, dtype, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status);`
- Ποιος το έστειλε το μήνυμα και ποια είναι η ετικέτα του;
  - `status->MPI_TAG`
  - `status->MPI_SOURCE`

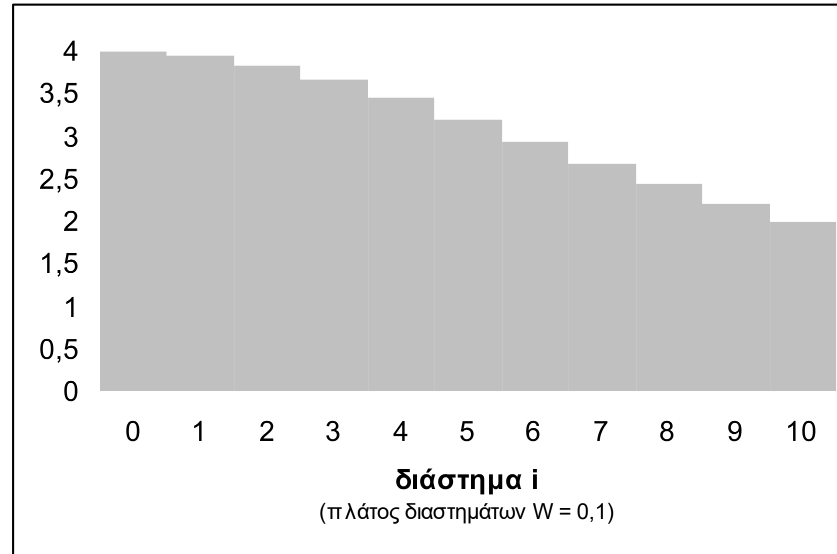
# Βασική τεχνική

- Οι διεργασίες καθορίζονται (και μοιράζονται τη δουλειά) όπως και στο μοντέλο κοινού χώρου διευθύνσεων
  - π.χ. μπορώ να κάνω διάσπαση βρόχου, διαχωρισμό σκακιέρας, αυτοδρομολόγηση κλπ.
- Επειδή, όμως, δεν υπάρχει τίποτε κοινό ανάμεσα στις διεργασίες, θα υπάρχει αναγκαστικά μία διεργασία η οποία:
  - αρχικοποιεί τις δομές
  - μοιράζει τα δεδομένα σε άλλες
  - συλλέγει τα επιμέρους αποτελέσματα
  - και ίσως τα δείχνει στον χρήστη

# Υπολογισμός του π = 3,14…

- Αριθμητική ολοκλήρωση

$$\pi = \int_0^1 \frac{4}{1+x^2}$$




$$\approx \sum_{i=0}^{N-1} \frac{4W}{1+\left[\left(i+{}^{1}/_{2}\right)W\right]^2}$$

# Υπολογισμός του π = 3,14… (I)

***Σειριακό πρόγραμμα***

```
int   i, N = 512;                        /* Ορισμός μεταβλητών */
float pi = 0.0, W = 1.0/N;

for (i = 0; i < N; i++)                  /* Ο υπολογισμός */
  pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
```

***Παραλληλοποίηση I:** μία διεργασία ανά επανάληψη (very fine grain)*

Η διεργασία i θα υπολογίσει τον i-οστό όρο του αθροίσματος:

```
  4*W / (1 + (i+0.5)*(i+0.5)*W*W);
```

***Παράλληλοποίηση II:** μία διεργασία ανά μπλοκ επαναλήψεων (courser grain)*

Κάθε διεργασία myid θα υπολογίσει WORK όρους του αθροίσματος

```
#define N     512      /* Όροι του αθροίσματος */
#define NPROC 32       /* Αριθμός επεξεργαστών/διεργασιών */
#define WORK  N/NPROC  /* Όροι ανά διεργασία */

for (i = 0; i < WORK; i++)                    /* Οι υπολογισμοί της διεργασίας */
  mysum += 4*W / (1 + ((myid*WORK+i)+0.5)*    /* (με τμηματική δρομολόγηση) */
                      ((myid*WORK+i)+0.5)*W*W);
```

PARALLEL PROCESSING GROUP

# Παράδειγμα: υπολογισμός του π με MPI

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  double     W, result = 0.0, temp;
  int        N, i, myid, nproc;
  MPI_status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Initialization */
  if (myid == 0) {
     printf("Enter number of divisions: ");
     scanf("%d", &N);
     for (i = 1; i < nproc; i++)
       MPI_Send(&N, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(&N, 1, MPI_INT, 0, 0,
             MPI_COMM_WORLD, &status);

  /* The actual computation */
  W = 1.0 / N;
  for (i = myid; i < N; i += nproc)
    result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);

  /* Gather results */
  if (myid == 0) {
    for (i = 1; i < nproc; i++) {
       MPI_Recv(&temp, 1, MPI_DOUBLE, i, 0,
                MPI_COMM_WORLD, &status);
       result += temp;
    }
    printf("pi = %lf\n", result);
  }
  else
    MPI_Send(&result, 1, MPI_DOUBLE, 0, 0,
             MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

# Υπολογισμός του π

- Βελτίωση: Παραλαβή μηνυμάτων όπως καταφθάνουν
  - Βελτίωση στην ταχύτητα, μείωση ανάγκης για buffering κλπ., *αρκεί να το επιτρέπει ο αλγόριθμος.*

```
/* Gather results */
if (myid == 0) {
  for (i = 1; i < nproc; i++) {
     MPI_Recv(&temp, 1, MPI_DOUBLE, i, 0,
              MPI_COMM_WORLD, &status);
     result += temp;
  }
  printf("pi = %lf\n", result);
}
```

```
/* Gather results */
if (myid == 0) {
  for (i = 1; i < nproc; i++) {
     MPI_Recv(&temp, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
              0, MPI_COMM_WORLD, &status);
     result += temp;
  }
  printf("pi = %lf\n", result);
}
```

# Παράδειγμα: πίνακας επί διάνυσμα

- Σειριακά:

```
float A[N][N], v[N], res;

for (i = 0; i < N; i++) {
  sum = 0.0;
  for (j = 0; j < N; j++)
    sum += A[i][j]*v[j];
  res[i] = sum;
}
```

- Θα χρησιμοποιήσουμε τμηματική δρομολόγηση στον βρόχο του i
- Όπως και στο μοντέλο κοινού χώρου διευθύνσεων, η κάθε διεργασία θα εκτελέσει το παρακάτω:

```
WORK = N / nprocs;   /* Στοιχεία ανά διεργασία */
sum = 0.0;

for (i = 0; i < WORK; i++)
{
  for (j = 0; j < N; j++)
    sum += A[myid*WORK+i][j]*v[j];
```

# Πίνακας επί διάνυσμα, συνέχεια

- Επίσης η διεργασία 0 θα «συλλέξει» όλα τα επιμέρους στοιχεία του αποτελέσματος από τις άλλες διεργασίες για να σχηματίσει την τελική απάντηση. Επομένως, κάθε διεργασία θα πρέπει στο τέλος να κάνει:

```
  /* Το sum έχει το (myid*WORK+i)-οστό στοιχείο του αποτελέσματος */
  MPI_Send(&sum, 1, MPI_FLOAT, 0, myid*WORK+i, MPI_COMM_WORLD);
}
```

- Τέλος, η διεργασία 0 θα πρέπει να κάνει την αρχικοποίηση, δηλαδή:
  - να αρχικοποιήσει (π.χ. να διαβάσει από το πληκτρολόγιο, από αρχείο) τα A[][] και v[].
  - Να τα στείλει σε όλες τις άλλες διεργασίες (υπενθύμιση: δεν υπάρχουν κοινές μεταβλητές!)

PARALLEL PROCESSING GROUP

# Τελικός κώδικας: πίνακας επί διάνυσμα (I)

```c
#define N 100         /* Η διάσταση του πίνακα */

void matrix_times_vector(int myid, int nproc) {
  int        i, j;
  int        WORK = N / nproc;     /* Rows per process */
  double     sum = 0.0, v[N];
  MPI_Status status;

  if (myid == 0) {                 /* Process 0 */
    double A[N][N], res[N];

    initialize_elements(A, v);     /* Some initalization */
    for (i = 1; i < nproc; i++) {  /* Send matrix & vector */
      MPI_Send(A[i*WORK], WORK*N, MPI_DOUBLE, i, 0,
               MPI_COMM_WORLD);
      MPI_Send(v, N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
    }
    for (i = 0; i < WORK; i++) {   /* Doing my own work */
      for (sum = 0.0, j = 0; j < N; j++)
        sum += A[i][j]*v[j];
      res[i] = sum;
    }
    /* Gather all result elements from other processes */
    for (i = WORK; i < WORK*nproc; i++) {
      MPI_Recv(&sum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
               MPI_COMM_WORLD, &status);
      res[ status.MPI_TAG ] = sum; /* Tag has elem position! */
    }
    show_result(res);              /* Display the end result */
  }
  else {                           /* All other processes */
    double *B = Balloc(WORK);

    MPI_Recv(B, WORK*N, MPI_DOUBLE, 0, 0,
             MPI_COMM_WORLD, &status);
    MPI_Recv(v, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
             &status);
    for (i = 0; i < WORK; i++) {
      for (sum = 0.0, j = 0; j < N; j++)
        sum += B[i*N+j]*v[j];

      /* sum has the (myid*WORK+i)-th elem of result */
      MPI_Send(&sum, 1, MPI_DOUBLE, 0, myid*WORK+i,
               MPI_COMM_WORLD);
    }
  }
}
```

# Βελτιστοποίηση

- Οι επικοινωνίες είναι ο εχθρός της ταχύτητας!
- Κοιτάμε να τις αποφεύγουμε όσο γίνεται.
- Καλύτερα λίγα και μεγάλα μηνύματα, παρά πολλά και μικρά.
  - Ομαδοποίηση μηνυμάτων όσο γίνεται.

**Διεργασίες εκτός της 0:**

```
else {        /* All other processes */
  double *B = Balloc(WORK);

  MPI_Recv(B, WORK*N, MPI_DOUBLE, 0, 0,
           MPI_COMM_WORLD, &status);
  MPI_Recv(v, N, MPI_DOUBLE, 0, 0,
           MPI_COMM_WORLD, &status);
  for (i = 0; i < WORK; i++) {
    for (sum = 0.0, j = 0; j < N; j++)
      sum += B[i*N+j]*v[j];
    mypart[i] = sum;     /* Keep element */
  }
  /* myid*WORK is the 1st computed element */
  MPI_Send(mypart, WORK, MPI_DOUBLE, 0, myid*WORK,
           MPI_COMM_WORLD);
}
```

**Διεργασία 0:**

```
if (myid == 0) {
  ...
  /* Gather results from other processes */
  for (i = 1; i < nproc; i++) {
    MPI_Recv(mypart, WORK, MPI_DOUBLE, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    for (j = 0; j < WORK; j++)    /* Place elements */
      res[ j + status.MPI_TAG ] = mypart[j];
  }
  show_result(res);       /* Display the end result */
}
```

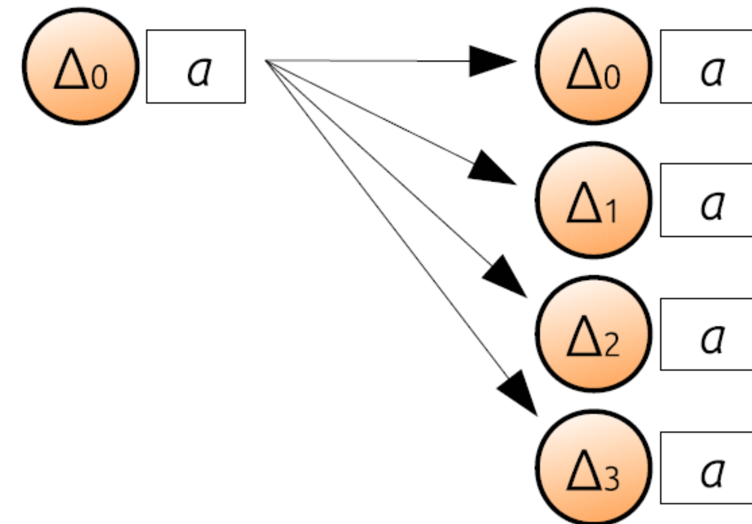
# Συλλογικές επικοινωνίες

Collective communications

# Συλλογικές επικοινωνίες (collective communications)

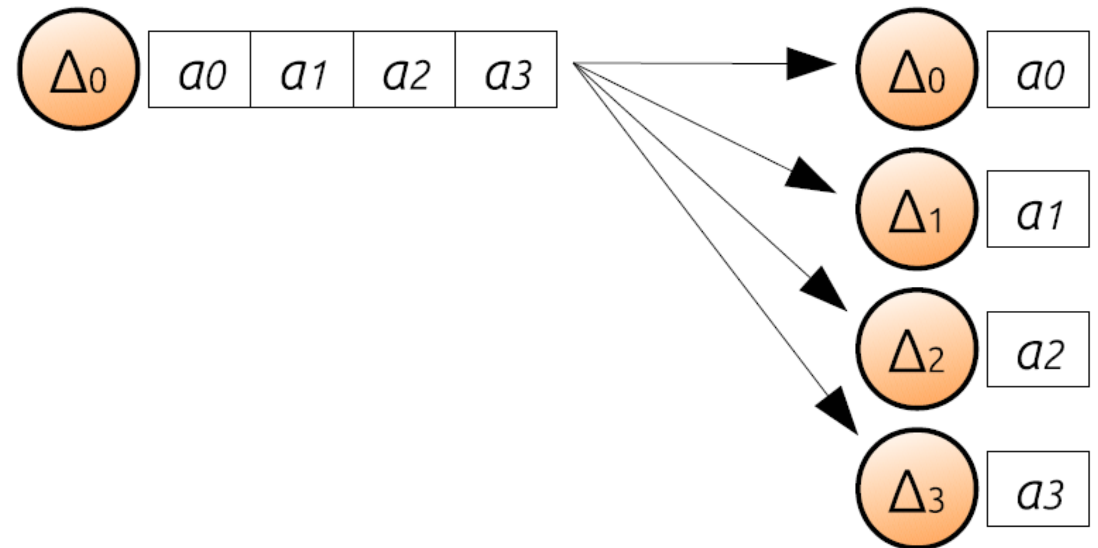
- Εκτός από την επικοινωνία ενός ζεύγους διεργασιών (“unicast communication”), υπάρχει πολύ συχνά ανάγκη για επικοινωνία μεταξύ όλων των διεργασιών μαζί.
  - “collective” communications
  - διάφορων ειδών

- Εκπομπή (broadcasting, one-to-all)
  - ίδιο μήνυμα από μία διεργασία προς όλες τις άλλες



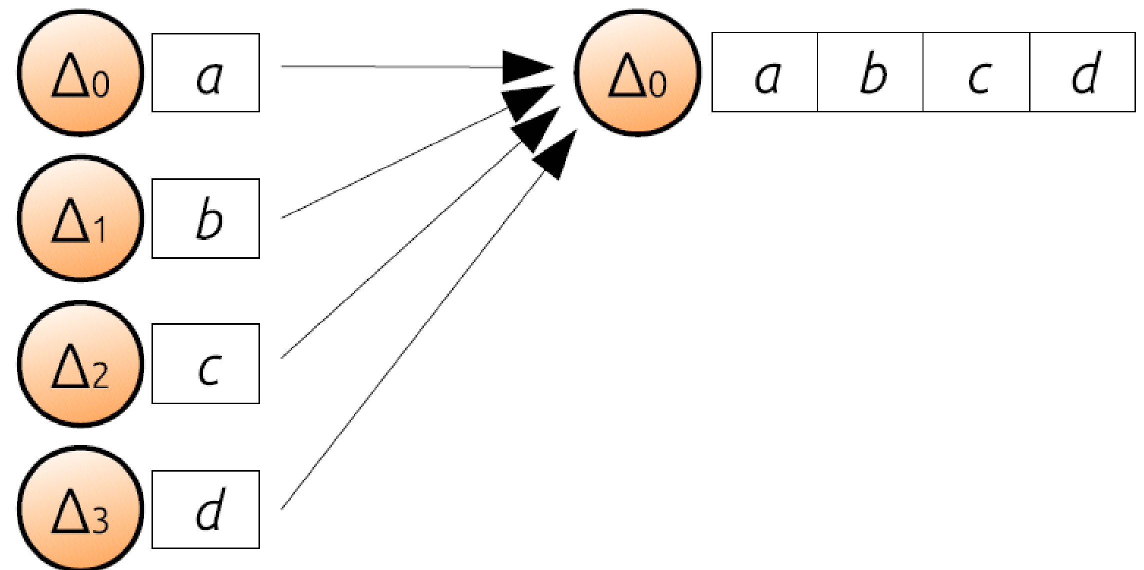
# Συλλογικές επικοινωνίες

- Διασκόρπιση (scattering, personalized one-to-all)
  - διαφορετικά μηνύματα από μία διεργασία προς όλες τις άλλες

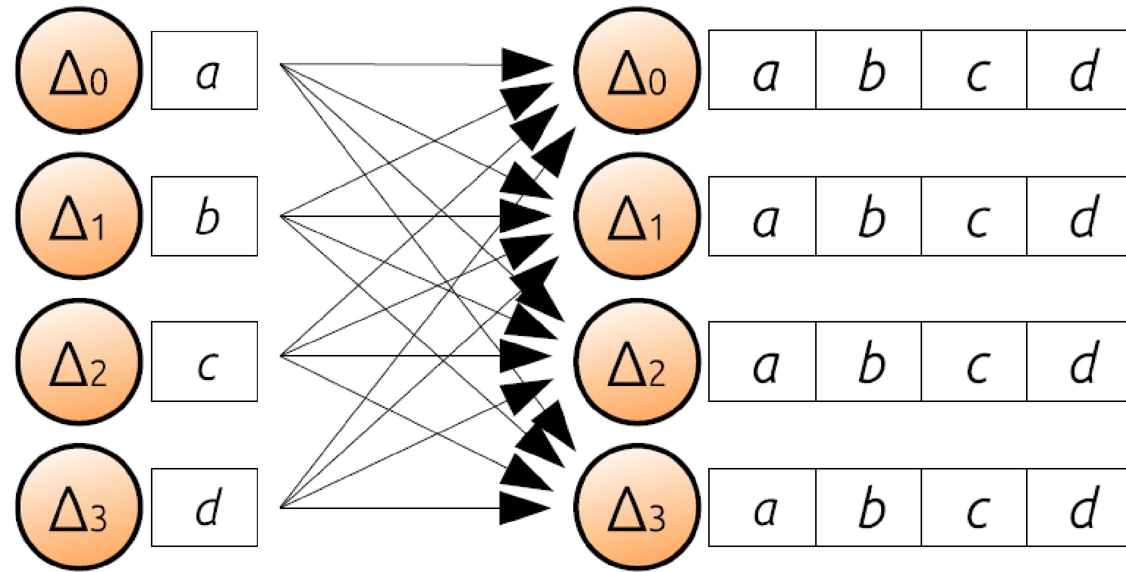


# Συλλογικές επικοινωνίες

- Συλλογή (gathering, personalized all-to-one)
  - ( αντίθετο της διασκόρπισης )
  - μία διεργασία λαμβάνει ένα μήνυμα από κάθε μία από τις υπόλοιπες

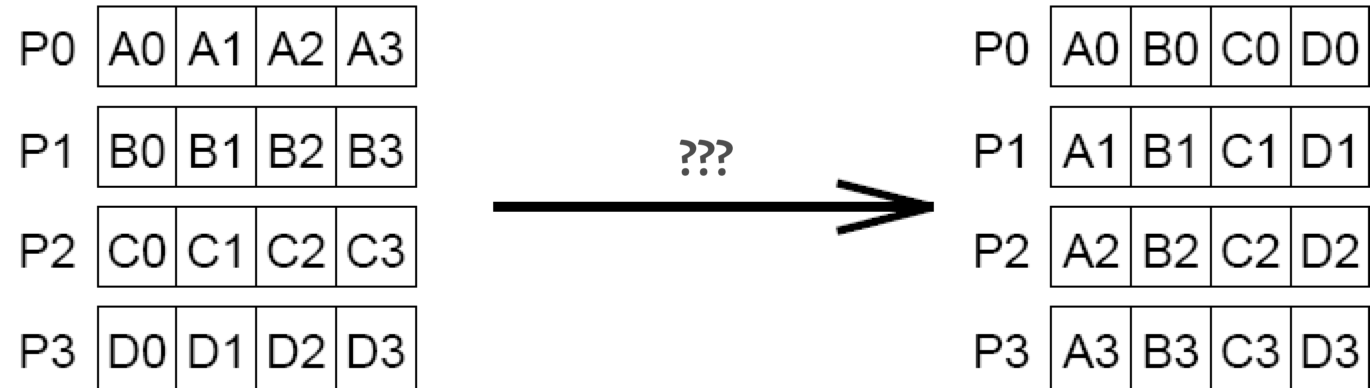


# Συλλογικές επικοινωνίες




- Πολλά πιθανά ονόματα:
  - πολλαπλή εκπομπή (multinode ή all-to-all broadcasting)
  - Όλες εκτελούν εκπομπή (ή όλες εκτελούν *το ίδιο* gather)
  - Στο MPI λέγεται **allgather**

# Συλλογικές επικοινωνίες

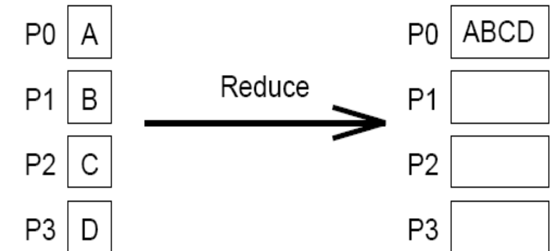


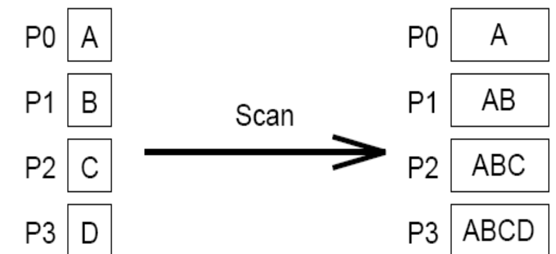

- Πιθανά ονόματα:
  - ολική ανταλλαγή (multinode gather/scatter ή all-to-all personalized ή total exchange)
  - ( όλες οι διεργασίες εκτελούν τη δική τους διασκόρπιση )

    κάθε διεργασία έχει διαφορετικό μήνυμα για κάθε άλλη
  - Στο MPI λέγεται **alltoall**

# Άλλες συλλογικές επικοινωνίες/υπολογισμοί

- Λειτουργίες υποβίβασης (reduction operations)
  - Από μέγεθος μεγαλύτερης διάστασης (π.χ. διάνυσμα) «υποβιβαζόμαστε» σε μέγεθος μικρότερης διάστασης (π.χ. βαθμωτό)
  - global sum, product, εύρεση max, min κλπ




- Κλήσης φραγής (barrier calls)
- Μερική εμπομπή (multicast)
  - Εκπομπή μόνο σε μερικές διεργασίες, όχι σε όλες
  - Είναι πιο «δύσκολη» επικοινωνία από την ολική εκπομπή (!)
- και άλλες ...

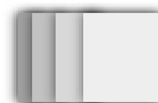
# Κλήσεις

- Την ίδια κλήση κάνουν *όλες οι διεργασίες*

- `MPI_Bcast(buf, n, dtype, rootrank,MPI_COMM_WORLD);`
  - το rootrank δηλώνει ποιος κάνει την εκπομπή

- ```
  MPI_Scatter(sbuf, sn, stype,
              rbuf, rn, rtype, rootrank, MPI_COMM_WORLD);
  ```
  - Μόνο για την πηγή μετράνε τα 3 πρώτα ορίσματα
  - Το sn είναι ο # στοιχείων που θα σταλεί σε κάθε διεργασία (**περιλαμβανομένης και της πηγής**) και πρέπει να είναι ίδιος με το rn.
  - Άρα αν υπάρχουν N διεργασίες, το sbuf πρέπει να έχει N*sn στοιχεία.
  - Κάθε διεργασία παραλαμβάνει τα στοιχεία της στο rbuf

- ```
  MPI_Gather(sbuf, sn, stype,
             rbuf, rn, rtype, targetrank, MPI_COMM_WORLD);
  ```
  - Μόνο για τη διεργασία-αποδέκτη μετράνε τα ορίσματα 4-6
  - Το sn είναι ο # στοιχείων που θα σταλεί η κάθε διεργασία (**περιλαμβανομένης και του αποδέκτη**).



# Κλήσεις, συνέχεια

- `MPI_Allgather(sbuf, sn, stype, rbuf, rn, rtype, MPI_COMM_WORLD);`
  - Ίδια με MPI_Gather() μόνο που όλες οι διεργασίες πρέπει να έχουν receive buffer
- `MPI_Alltoall(sbuf, sn, stype, rbuf, rn, rtype, MPI_COMM_WORLD);`
  - Παρόμοιες παράμετροι με με MPI_Allgather()
- `MPI_Reduce(sbuf, rbuf, n, dtype, op, targetrank, MPI_COMM_WORLD);`
  - το op είναι `MPI_MAX/MIN/SUM/PROD/LAND/BAND/LOR/BOR/LXOR/BXOR` (αν και μπορεί κανείς να ορίσει και δικά του)
  - τέλος υπάρχουν και τα MPI_MINLOC και MPI_MAXLOC που μαζί με το min/max επιστρέφουν το rank της διεργασίας που το διαθέτει
  - **Όλες** οι διεργασίες πρέπει να διαθέτουν send & receive buffer (sbuf & rbuf)
  - Η λειτουργία, αν το n είναι > 1, γίνεται σε κάθε στοιχείο ξεχωριστά
- `MPI_Allreduce(sbuf, rbuf, n, dtype, op, MPI_COMM_WORLD);`
  - Ίδια με MPI_Reduce() μόνο που όλες οι διεργασίες παίρνουν το αποτέλεσμα
- και άλλες…

# Παράδειγμα: υπολογισμός του π με MPI

```c
#include <mpi.h>

int main(int argc, char *argv[])
{
  double     W, result = 0.0, temp;
  int        N, i, myid, nproc;
  MPI_status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Initialization */
  if (myid == 0) {
     printf("Enter number of divisions: ");
     scanf("%d", &N);
     for (i = 1; i < nproc; i++)
       MPI_Send(&N, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(&N, 1, MPI_INT, 0, 0,
             MPI_COMM_WORLD, &status);

  /* The actual computation */
  W = 1.0 / N;
  for (i = myid; i < N; i += nproc)
    result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);

  /* Gather results */
  if (myid == 0) {
    for (i = 1; i < nproc; i++) {
       MPI_Recv(&temp, 1, MPI_DOUBLE, i, 0,
                MPI_COMM_WORLD, &status);
       result += temp;
    }
    printf("pi = %lf\n", result);
  }
  else
    MPI_Send(&result, 1, MPI_DOUBLE, 0, 0,
             MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

# Παράδειγμα: υπολογισμός του π με MPI

```c
#include <mpi.h>

int main(int argc, char *argv[])
{
  double     W, result = 0.0, temp;
  int        N, i, myid, nproc;
  MPI_status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Initialization */
  if (myid == 0) {
     printf("Enter number of divisions: ");
     scanf("%d", &N);
     for (i = 1; i < nproc; i++)
       MPI_Send(&N, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(&N, 1, MPI_INT, 0, 0,
             MPI_COMM_WORLD, &status);

  /* The actual computation */
  W = 1.0 / N;
  for (i = myid; i < N; i += nproc)
    result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);

  /* Gather results */
  if (myid == 0) {
    for (i = 1; i < nproc; i++) {
       MPI_Recv(&temp, 1, MPI_DOUBLE, i, 0,
                MPI_COMM_WORLD, &status);
       result += temp;
    }
    printf("pi = %lf\n", result);
  }
  else
    MPI_Send(&result, 1, MPI_DOUBLE, 0, 0,
             MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

# Παράδειγμα: υπολογισμός του π με MPI

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  double     W, result = 0.0, temp;
  int        N, i, myid, nproc;
  MPI_status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Initialization */
  if (myid == 0) {
     printf("Enter number of divisions: ");
     scanf("%d", &N);
  }
  MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

  /* The actual computation */
  W = 1.0 / N;
  for (i = myid; i < N; i += nproc)
    result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);

  /* Gather results */
  if (myid == 0) {
    for (i = 1; i < nproc; i++) {
       MPI_Recv(&temp, 1, MPI_DOUBLE, i, 0,
                MPI_COMM_WORLD, &status);
       result += temp;
    }
    printf("pi = %lf\n", result);
  }
  else
    MPI_Send(&result, 1, MPI_DOUBLE, 0, 0,
             MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

# Παράδειγμα: υπολογισμός του π με MPI

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  double     W, result = 0.0, temp, pi = 0.0;
  int        N, i, myid, nproc;
  MPI_status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Initialization */
  if (myid == 0) {
     printf("Enter number of divisions: ");
     scanf("%d", &N);
  }
  MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

  /* The actual computation */
  W = 1.0 / N;
  for (i = myid; i < N; i += nproc)
    result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);

  /* Sum result */
  MPI_Reduce(&result, &pi, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);

  if (myid == 0)
    printf("pi = %lf\n", result);

  MPI_Finalize();
  return 0;
}
```

# Παράδειγμα: πίνακας επί διάνυσμα (ΙΙ) – συλλογικά

```
void matrix_times_vector(int myid, int nproc) {
  int       i, j;
  int       WORK = N / nproc;    /* Rows per process */
  double    sum, v[N], A[N][N], res[N], *mypart, (*B)[N];

  if (myid == 0)
    initialize_elements(A, v);

  B = allocrows(WORK);           /* Allocate space for my rows.. */
  mypart = allocvector(WORK);    /* ..and for my results */

  MPI_Scatter(A, WORK*N, MPI_DOUBLE, B, WORK*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(v, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  for (i = 0; i < WORK; i++) {
    for (sum = 0.0, j = 0; j < N; j++)
      sum += B[i][j]*v[j];
    mypart[i] = sum;
  }

  MPI_Gather(mypart, WORK, MPI_DOUBLE, res, WORK, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  if (myid == 0)
    show_result(res);
}
```

# Γιατί συλλογικές επικοινωνίες;

1. Μείωση του κώδικα
   - Πολύ λιγότερες γραμμές σε μερικές περιπτώσεις
2. Βελτίωση του κώδικα
   - Καλύτερη αναγνωσιμότητα
   - Ευκολότερη κατανόηση
   - Απλούστερη συντήρηση και αποσφαλμάτωση
3. Βελτίωση των επιδόσεων
   - Η υλοποίηση της βιβλιοθήκης του MPI εκμεταλλεύεται το σύστημα
   - Ανάλογα με την τοπολογία, οι συλλογικές επικοινωνίες δρομολογούνται επάνω σε μονοπάτια/δέντρα που μειώνουν πάρα πολύ τους χρόνους μετάδοσης σε σχέση με ιδιωτικά μηνύματα μεταξύ ζευγών διεργασιών.

Ισχυρό προγραμματιστικό «εργαλείο» για μεταβίβαση μηνυμάτων.

# Ασφαλείς επικοινωνίες

# Προβλήματα;

- Δύο διεργασίες ανταλλάσουν δεδομένα:

  P0:
  ```
  MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(&b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
  ```
  P1:
  ```
  MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  ```
  - Μία χαρά!

- Το επόμενο;

  P0:
  ```
  MPI_Recv(&b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
  MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  ```
  P1:
  ```
  MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  ```
  - deadlock!

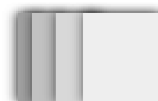
# Ασφάλεια

- Αυτό;

P0:
```
MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
MPI_Recv(&b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
```
P1:
```
MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

  - Είναι εντάξει ΜΟΝΟ εφόσον υπάρχουν αρκετοί buffers (στη μεριά του παραλήπτη) να φυλάνε τα μηνύματα που στάλθηκαν (μέχρι να την οριστική παραλαβή τους)
  - Αν δεν υπάρχει χώρος, τότε το MPI «μπλοκάρει» μέχρι να βρεθεί χώρος στον παραλήπτη
  - Αν οι buffers και των δύο διεργασιών είναι «γεμάτοι», τότε DEADLOCK!
  - Άρα η ορθότητα / ασφάλεια του κώδικα εξαρτάται από την ποσότητα των διαθέσιμων buffers
    - ΜΗ ΑΣΦΑΛΕΣ ΠΡΟΓΡΑΜΜΑ

- Παρόμοια περίπτωση είναι και όταν N διεργασίες ανταλλάσουν κυκλικά από ένα δεδομένο (η i στέλνει στην i+1 και λαμβάνει από την i-1). Ποιά είναι η ασφαλέστερη υλοποίηση;
  - Οι άρτιες διεργασίες αρχικά λαμβάνουν και στη συνέχεια στέλνουν
  - Οι περιττές κάνουν το ανάποδο
  - *odd-even rule*



# Συναρτήσεις που παρέχουν ασφάλεια

- Όταν κάθε διεργασία στέλνει & λαμβάνει, το MPI παρέχει μία συνάρτηση ώστε να μην τίθεται θέμα ασφάλειας:

```
MPI_Sendrecv(sbuf, sn, sdtype, torank, stag,
             rbuf, rn, rdtype, fromrank, rtag,
             MPI_COMM_WORLD, status);
```

- Αυτή η συνάρτηση μπορεί επίσης να παραλάβει και μηνύματα που στάλθηκαν με απλό MPI_Send(), ενώ το μήνυμα που στέλνεται από αυτήν μπορεί να παραληφθεί και με απλό MPI_Recv()

- Αν θέλουμε ο buffer αποστολής & λήψης να είναι ο ίδιος, τότε:

```
MPI_Sendrecv_replace(buf, n, dtype, torank, stag, fromrank, rtag,
                     MPI_COMM_WORLD, status);
```

# MPI – προχωρημένες λειτουργίες

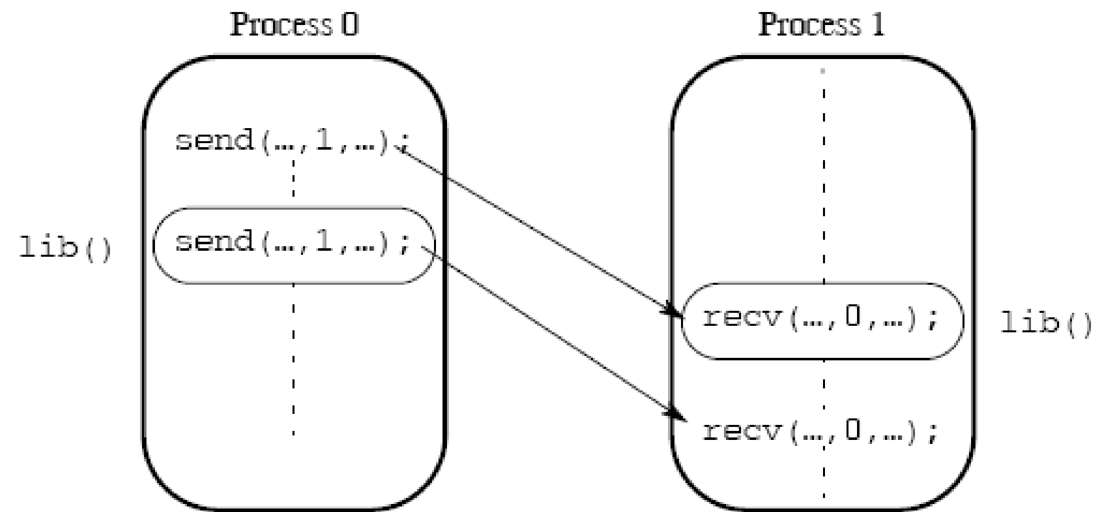
# Communicators

- Αν θέλω να ορίσω ομάδα (group) διεργασιών τότε δημιουργώ έναν communicator και οι επικοινωνίες που γίνονται με αυτόν αφορούν μόνο διεργασίες του γκρουπ.
  - Ο MPI_COMM_WORLD αφορά το γκρουπ ΟΛΩΝ των διεργασιών που υπάρχουν.
  - Σε όλες τις κλήσεις που είδαμε, μπορεί κανείς να αντικαταστήσει το MPI_COMM_WORLD με όποιον άλλον δικόν του έχει δημιουργήσει – τα send/receive με άλλον communicator απευθύνονται μόνο στις διεργασίες του αντίστοιχου γκρουπ.
  - Σε κάθε γκρουπ/communicator, οι διεργασίες έχουν ακολουθιακό rank.
  - Κάθε διεργασία ανήκει στο MPI_COMM_WORLD και πιθανώς σε πολλά άλλα communicators
  - Αν μία διεργασία ανήκει σε έναν communicator comm, τότε δεν σημαίνει ότι τα myid1 και myid2 είναι οπωσδήποτε ίδια:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid1);
MPI_Comm_rank(comm, &myid2);
```

# Άλλη μία ανάγκη …

- Εκτός από την μεριά του χρήστη, υπάρχει και άλλη μία ανάγκη για communicators:
  - Το MPI εγγυάται ότι τα μηνύματα φτάνουν στην διεργασία-παραλήπτη, με τη σειρά που στάλθηκαν από τη διεργασία-αποστολέα.
  - Και οι δύο διεργασίες χρησιμοποιούν μία βιβλιοθήκη η οποία στέλνει/λαμβάνει τα δικά της μηνύματα. Φανταστείτε ότι ο αποστολέας κάνει μία κλήση σε αυτήν *μετά το send* ενώ ο παραλήπτης κάτι χρειάστηκε από τη βιβλιοθήκη *πριν το receive*. Τι θα γίνει;



## … βιβλιοθήκες που δεν ελέγχονται από τον χρήστη

- Δεν μπορείς να βασιστείς στα tags διότι δεν γνωρίζεις τι tags χρησιμοποιεί η βιβλιοθήκη.
- Επίσης δεν μπορείς να βασιστείς στα ranks μιας και οι διεργασίες καλούν συναρτήσεις της βιβλιοθήκης (η βιβλιοθήκη δεν είναι ξεχωριστή διεργασία – είναι απλά ένα σύνολο ρουτινών)

- Λύση: communicators
  - Οι βιβλιοθήκες, για αυτούς τους λόγους, εσωτερικά δημιουργούν «ιδιωτικό» communicator και όλες οι αποστολές/λήψεις γίνονται μέσω αυτού
  - Έτσι δεν υπάρχει περίπτωση να «μπερδευτούν» με τα μηνύματα των διεργασιών του χρήστη.

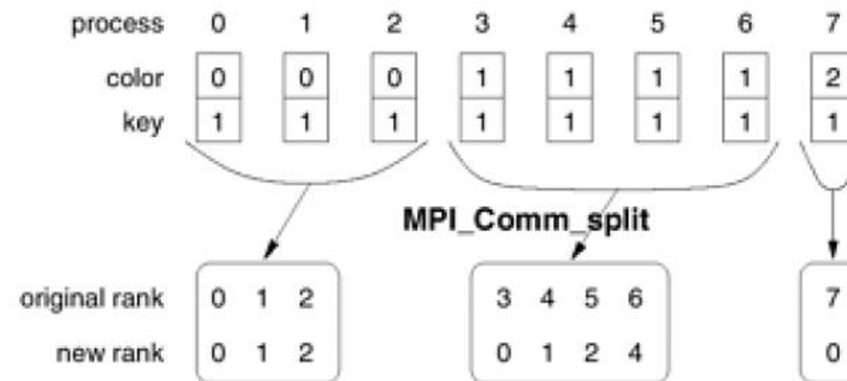
# Φτιάχνοντας communicators

- Νέος communicator φτιάχνεται (αυτόματα) όταν δημιουργούνται τοπολογίες (παρακάτω…)
- Ένας άλλος τρόπος είναι να «χωρίσεις» τις διεργασίες από έναν υπάρχοντα communicator:

```
MPI_Comm_split(oldcomm, int color, int key, &newcomm);
```

Π.χ. `MPI_Comm_split(MPI_COMM_WORLD, int color, int key, &newcomm);`

  - Είναι *συλλογική διαδικασία*. Πρέπει ΟΛΕΣ οι διεργασίες του `oldcomm` να την καλέσουν!
  - Όσες διεργασίες δώσουν το ίδιο color θα είναι στον ίδιο νέο communicator
    - Άρα φτιάχνονται τόσοι νέοι communicators όσα και τα διαφορετικά colors
  - Το rank κάθε διεργασίας στον νέο communicator που θα ανήκει καθορίζεται από το key. Σε ισοπαλία, χρησιμοποιείται η κατάταξη στον παλιό communicator



# Τοπολογίες διεργασιών

- Κανονικά οι διεργασίες αριθμούνται ακολουθιακά (γραμμικά) 0, 1, … N-1.

- Πολλές φορές θέλουμε μία διαφορετική αρίθμηση
  - Στο MPI μπορούμε να ορίσουμε εικονικές «τοπολογίες»
  - Η κάθε διεργασία έχει τη δική της αρίθμηση (ετικέτα) στην κάθε τοπολογία
  - Τοπολογίες καρτεσιανού γινομένου μόνο (πλέγματα και tori)
  - Η νέα τοπολογία ΔΕΝ είναι πλέον αυτή του MPI_COMM_WORLD!
    - Ορίζεται ένας νέος “communicator” για την τοπολογία αυτή

  - Επίσης, μπορεί το MPI να τις χρησιμοποιήσει ώστε να κάνει καλύτερο mapping με την τοπολογία του δικτύου (ο χρήστης δεν έχει έλεγχο πάνω σε αυτή την αντιστοίχιση)

# Δημιουργία καρτεσιανών τοπολογιών

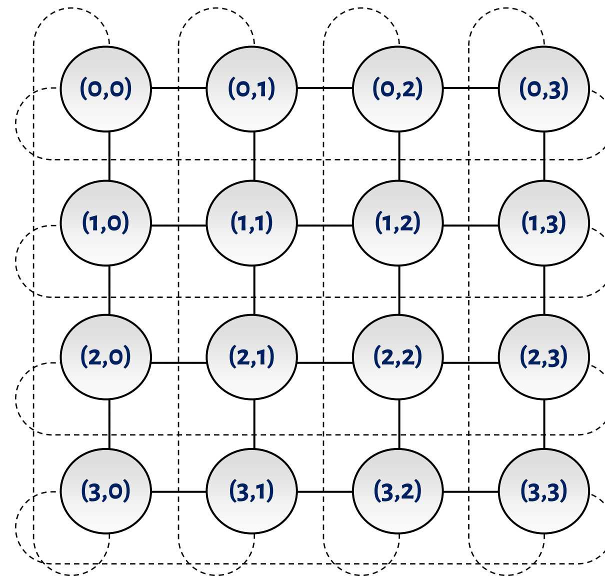
- Δημιουργούμε καρτεσιανές τοπολογίες (πλέγματα) με:

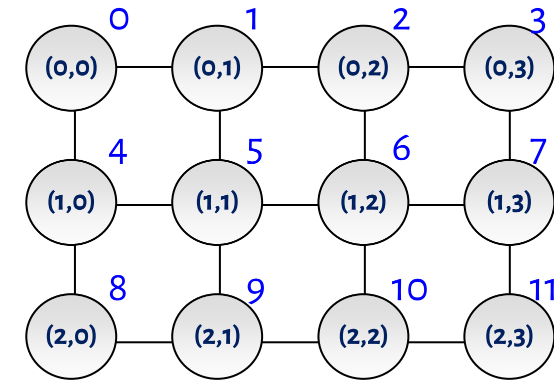
```
MPI_Cart_create(MPI_Comm comm_old, int ndims,
        int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
```

  Οι διεργασίες που ανήκουν στον παλιό communicator δημιουργούν ένα νέο communicator με τοπολογία πλέγματος “ndims” διαστάσεων.
  - Αν το periods[i] είναι 1, τότε η διάσταση i θα είναι δακτύλιος (κύκλος)
  - Αν το reorder είναι 0, οι διεργασίες θα κρατήσουν το ίδιο (ακολουθιακό) rank που είχαν και στον παλιό communicator.

- Κάθε διεργασία θα έχει ως «ταυτότητα» μία ndims-άδα, δηλαδή ένα διάνυσμα ndims στοιχείων («συντεταγμένες» σε κάθε διάσταση).
  - έχοντας φυσικά και το κλασικό ακολουθιακό ranks της

# Τοπολογίες




(με μπλε τα ranks)

ndims =2

dims[0]=4, dims[1]=4

periods[0]=1, periods[1]=1

ndims =2

dims[0]=3, dims[1]=4

periods[0]=0, periods[1]=0

# Χρησιμοποιώντας καρτεσιανές τοπολογίες

- Μιας και τα send/receive εξακολουθούν να απαιτούν ranks ως ορίσματα, το MPI παρέχει ρουτίνες μετατροπής μεταξύ ranks και πολυδιάστατων συντεταγμένων:

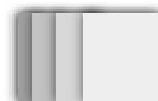
```
MPI_Cart_coords(MPI_Comm cartcomm, int rank, int ndims, int *coords)
MPI_Cart_rank(MPI_Comm cartcomm, int *coords, int *rank)
```

  – για «περιοδικές» διαστάσεις, αν η συντεταγμένη είναι εκτός ορίων, υπολογίζεται modulo στο μέγεθος της διάστασης

- Από τις πιο συνηθισμένες ενέργειες σε καρτεσιανές τοπολογίες είναι η κυκλική μεταφορά δεδομένων (shift). Κάθε διεργασία μπορεί να υπολογίσει άμεσα τον προορισμό και την πηγή που την αφορούν με:

```
MPI_Cart_shift(MPI_Comm cartcomm, int dir, int s_step,
      int *rank_source, int *rank_dest)
```

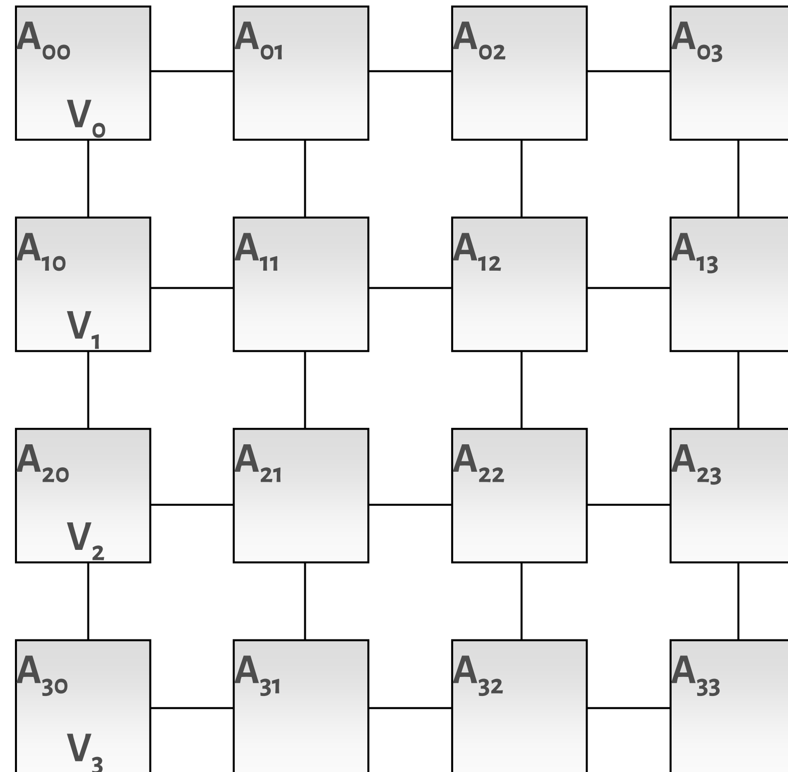
  – Το “direction” είναι η διάσταση στην οποία θα γίνει το shift (αρίθμηση από το 0). Το “s_step” είναι το μέγεθος (# θέσεων) του shift (θετικό ή αρνητικό). *Τα αποτελέσματα μπορούν να χρησιμοποιηθούν άμεσα σε MPI_Sendrecv().*



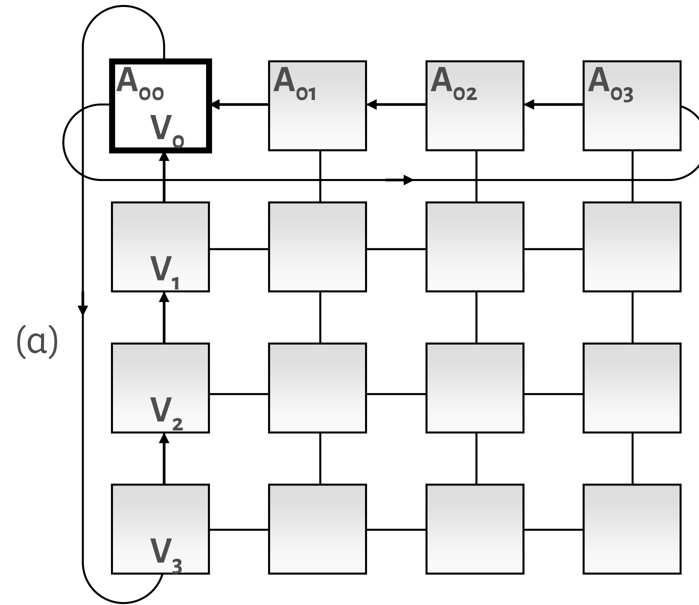
# Εφαρμογή: ο αλγόριθμος του Cannon για πολλαπλασιασμό πινάκων

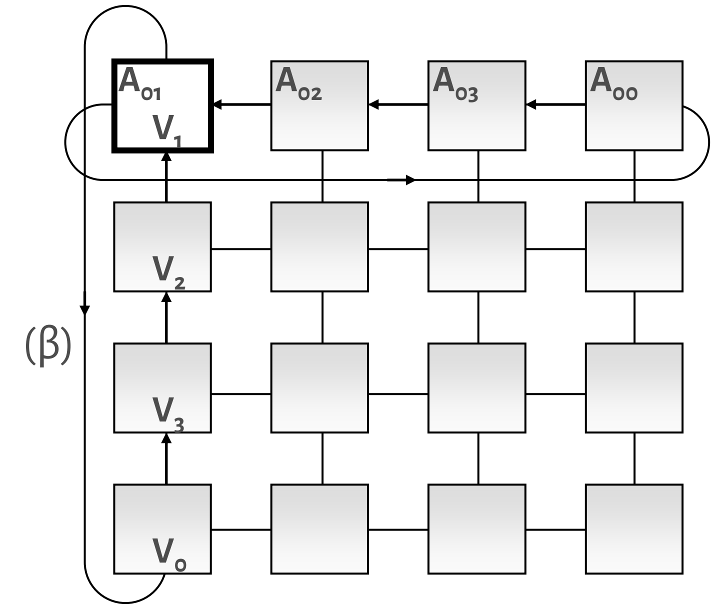
# Πολλαπλασιασμός A*v

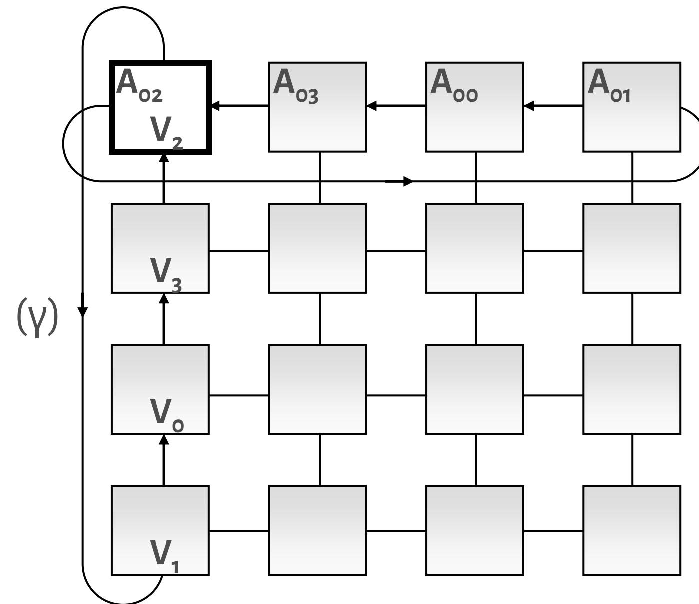
- Αρχική τοποθέτηση στοιχείων (πλέγμα/torus)

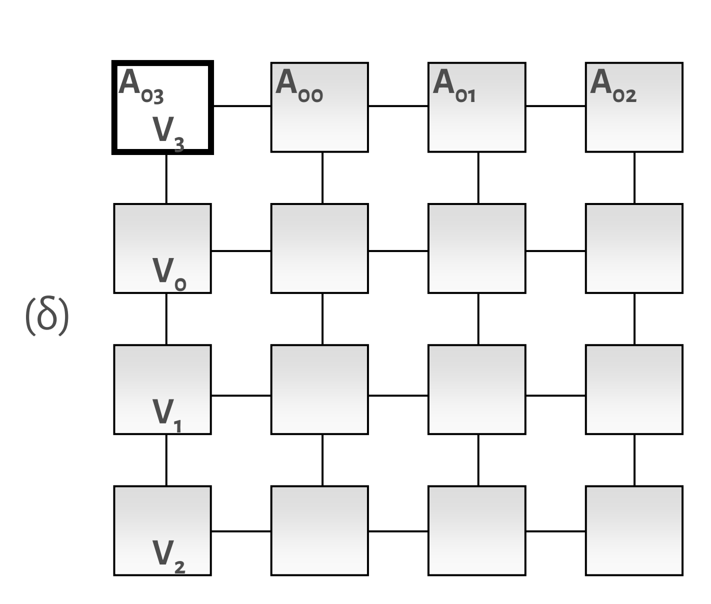


# Υπολογισμός 1ου στοιχείου του A*v



# Τι λείπει για να υπολογιστεί το 2ο στοιχείο;

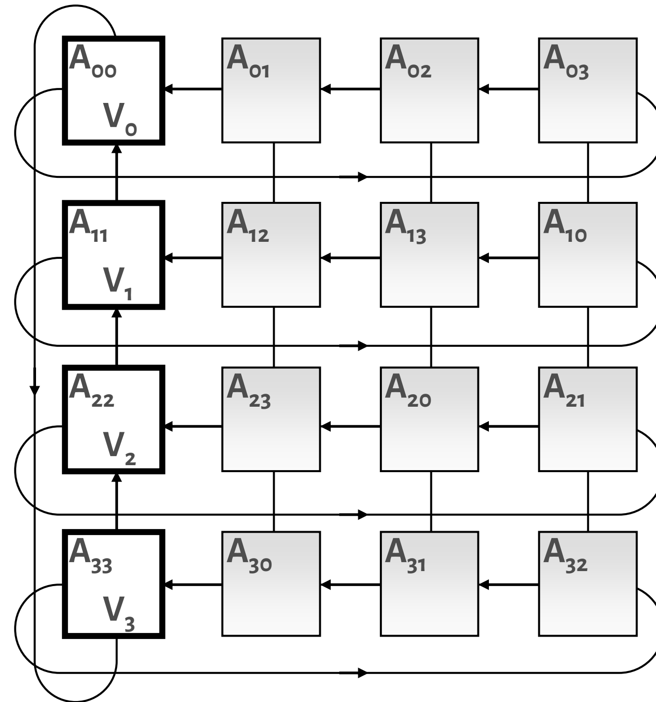
- Να γίνει το αντίστοιχο με την γραμμή 1 του A
- Όμως πρέπει να ξεκινήσουμε από το A[1][1]*v[1], μιας και το v[1] είναι αυτό που υπάρχει στην γραμμή 1. Πώς;
  - Απάντηση: *πριν* ξεκινήσει ο αλγόριθμος, κάνουμε τη γραμμή 1 SHIFT (rotate) 1 θέση προς τα αριστερά. Από εκεί και ύστερα, ο υπολογισμός του v[1] γίνεται ταυτόχρονα με αυτόν του v[0]!

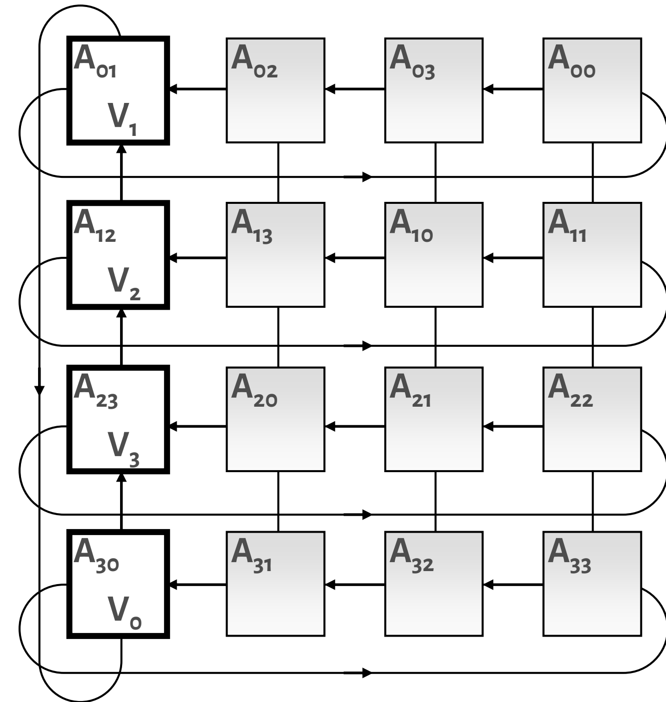
- Γενικά:
  - Φάση 1: προετοιμασία
    - Η γραμμή i ολισθάνει i θέσεις αριστερά (ώστε το A[i][i] να βρεθεί μαζί με το v[i])
  - Φάση 2: υπολογισμός

    LOOP (N φορές):

    1. Σε κάθε επεξεργαστή (i, 0) υπολογίζεται το γινόμενο του υπάρχοντος στοιχείου του πίνακα A με το υπάρχον στοιχείο του διανύσματος v.
    2. Ολίσθηση του διανύσματος v προς τα πάνω
    3. Ολίσθηση κάθε γραμμής του A προς τα αριστερά

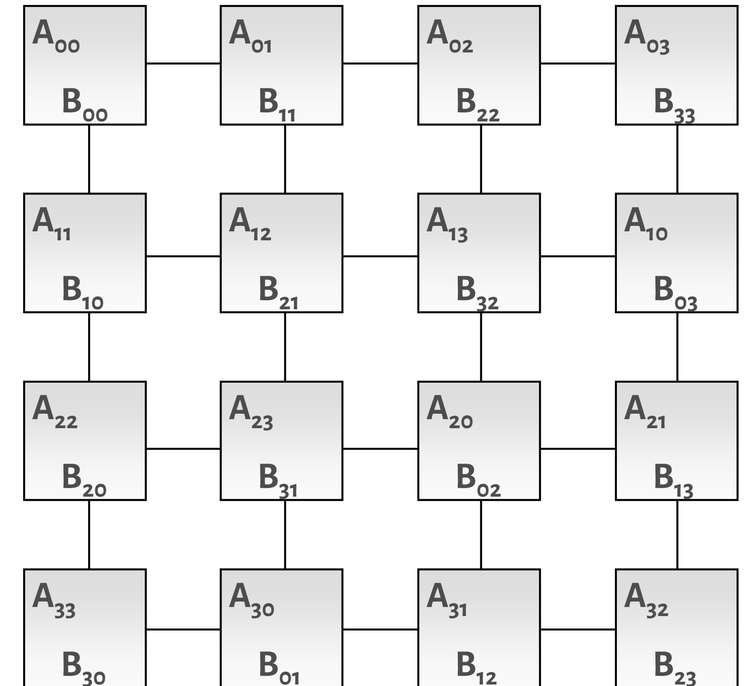
# A*v – αρχικές τοποθετήσεις & 1ο βήμα



# Γενίκευση σε A*B (αλγόριθμος Cannon)

- Αρχικά,
  - κάθε γραμμή *i* του πίνακα A ολισθαίνει κατά *i* θέσεις *αριστερά* και
  - κάθε στήλη *j* του πίνακα B ολισθαίνει κατά *j* θέσεις προς τα *πάνω*
- Στη συνέχεια γίνονται *n* επαναλήψεις, όπου σε κάθε επανάληψη, κάθε επεξεργαστής
  - πολλαπλασιάζει τα δύο στοιχεία που διαθέτει, αθροίζει και
  - κάθε γραμμή (στήλη) ολισθαίνει προς τα αριστερά (πάνω).
- Τελικά στον επεξεργαστή (*i*, *j*) θα βρεθεί το στοιχείο $C_{ij}$ του αποτελέσματος.



# Υλοποίηση σε MPI

- Τοπολογία πλέγματος 2D απαραίτητη για τον εύκολο χειρισμό
- Χρησιμοποιεί blocks του πίνακα (όχι απλά στοιχεία). Δηλαδή η κάθε διεργασία σε κάθε βήμα δεν πολλαπλασιάζει ένα στοιχείο του A με ένα στοιχείο του B αλλά έναν υποπίνακα του A με έναν υποπίνακα του B (δεν αλλάζει σε κάτι ο αλγόριθμος)
- Στην επόμενη σελίδα
  - Τα a και b είναι οι υποπίνακες που διαθέτει αρχικά κάθε διεργασία

```c
MatrixMatrixMultiply(int n, double *a, double *b, double *c, MPI_Comm comm)
{
  int i, nlocal;
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);

  /* Get the rank and coordinates in he new topology */
  MPI_Comm_rank(comm_2d, &my2drank);
  MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

  /* Compute ranks of the up and left shifts */
  MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
  MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

  /* Determine the dimension of the local matrix block */
  nlocal = n/dims[0];

  /* Perform the initial matrix alignment B */
  MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);
  MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);

  /* Get into the main computation loop */
  for (i=0; i<dims[0]; i++) {
    MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
          leftrank, 1, rightrank, 1, comm_2d, &status);
    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
          uprank, 1, downrank, 1, comm_2d, &status);
  }

  /* Restore the original distribution of a and b */
  MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);
  MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);

  MPI_Comm_free(&comm_2d); /* Free up communicator */
}
```

```c
MatrixMultiply(int n, double *a, double *b, double *c) { /* matrix-matrix multiplication c = a*b */
  int i, j, k;
  for (i=0; i<n; i++) for (j=0; j<n; j++) for (k=0; k<n; k++) c[i*n+j] += a[i*n+k]*b[k*n+j];
}
```

# Μη εμποδιστικές (non-blocking) επικοινωνίες

# Λήψη

- Μέχρι τώρα υποθέσαμε ότι μια διεργασία που κάνει MPI_Recv() διεργασία «κολλάει» και περιμένει μέχρι να έρθει το μήνυμα (blocking)

- Στην μη εμποδιστική λήψη:

```
MPI_Irecv(buf, m, dtype, frank, tag, comm, &status, &req);
```

  - Παραλαβή μόνο εάν έχει έρθει το μήνυμα
  - Αλλιώς *άμεση επιστροφή*

- Πρέπει να γνωρίζουμε αν παραλήφθηκε μήνυμα ή όχι!

```
MPI_Test(&req, &flag, &status);
```

  - Στο flag επιστρέφεται 0 (false) αν όχι, αλλιώς 1.
  - Προφανώς ο έλεγχος πρέπει να μπει σε κάποιο loop μέχρι να παραληφθεί το μήνυμα.
  - Για αναμονή (block) μέχρι να έρθει το μήνυμα:

```
MPI_Wait(&req, &status);
```

# Αποστολή

- Υπάρχει και εδώ η έννοια της εμποδιστικότητας αλλά όχι τόσο εμφανώς (δεν μιλάμε για συγχρονισμένες επικοινωνίες τύπου rendezvous όπου ο αποστολέας περιμένει μέχρι ο παραλήπτης να κάνει receive)
  - Η MPI_Send() επιστρέφει μόλις φύγει και το τελευταίο byte από τον κόμβο.
  - Δηλαδή, περιμένει μέχρι να ολοκληρωθεί η εξής διαδικασία:
    - Η βιβλιοθήκη MPI παίρνει το μήνυμα και το τοποθετεί πιθανώς σε δικούς της buffers
    - Το λειτουργικό σύστημα αντιγράφει πιθανώς από τον χώρο της βιβλιοθήκης σε buffers στον χώρο του πυρήνα (kernel space)
    - Από εκεί μεταφέρεται στους buffers του υποσυστήματος επικοινωνιών (π.χ. κάρτας δικτύου)
    - Από τους buffers της κάρτας «βγαίνουν» όλα τα bytes στο καλώδιο και ταξιδεύουν προς τον προορισμό

- Στη μη εμποδιστική αποστολή:

```
MPI_Isend(buf, n, dtype, trank, tag, comm, &req);
```

  - Επιστρέφει αμέσως
  - Επομένως, η μεταφορά από το `buf` στους διάφορους ενδιάμεσους buffers του τοπικού κόμβου ίσως να μην έχει ολοκληρωθεί!
  - *Τροποποίηση «στο καπάκι» του* `buf` *μπορεί να καταστρέψει το προηγούμενο μήνυμα που ίσως δεν έχει προλάβει ακόμα να φύγει*
  - Πρέπει να σιγουρευτούμε ότι το MPI έχει πάρει τα δεδομένα από το `buf` πριν τον επαναχρησιμοποιήσουμε:

```
MPI_Test(&req, &flag, &status);
MPI_Wait(&req, &status);
```

# Γιατί μη εμποδιστικές επικοινωνίες;

- Κυρίως για βελτίωση επιδόσεων
  - επικάλυψη υπολογισμών και επικοινωνιών
  - *εκκίνηση όσο πιο **νωρίς** γίνεται*
    - ξεκίνα το send μόλις έχεις τα δεδομένα, ξεκίνα το receive μόλις έχεις άδειο buffer
  - *και ολοκλήρωση όσο πιο **αργά** γίνεται*
    - ολοκλήρωση του send μόλις είναι να ξαναστείλεις κάτι, ολοκλήρωση του receive μόλις είναι να χρησιμοποιήσεις τα δεδομένα

- Δευτερευόντως λόγων μείωσης πιθανότητας deadlock

P0:
```
MPI_Irecv(&b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status, &req);
MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```
P1:
```
MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

  - Δεν έχει deadlock!

```c
MatrixMatrixMultiply(int n, double *a, double *b, double *c, MPI_Comm comm)
{
  int i, nlocal;
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);

  /* Get the rank and coordinates in he new topology */
  MPI_Comm_rank(comm_2d, &my2drank);
  MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

  /* Compute ranks of the up and left shifts */
  MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
  MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

  /* Determine the dimension of the local matrix block */
  nlocal = n/dims[0];

  /* Perform the initial matrix alignment B */
  MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);
  MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);

  /* Get into the main computation loop */
  for (i=0; i<dims[0]; i++) {
    MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
           leftrank, 1, rightrank, 1, comm_2d, &status);
    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
           uprank, 1, downrank, 1, comm_2d, &status);
  }

  /* Restore the original distribution of a and b */
  MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);
  MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);

  MPI_Comm_free(&comm_2d); /* Free up communicator */
}
```

```c
MatrixMultiply(int n, double *a, double *b, double *c) { /* matrix-matrix multiplication c = a*b */
  int i, j, k;
  for (i=0; i<n; i++) for (j=0; j<n; j++) for (k=0; k<n; k++) c[i*n+j] += a[i*n+k]*b[k*n+j];
}
```

```c
MatrixMatrixMultiply(int n, double *a, double *b, double *c, MPI_Comm comm)
{
  int i, nlocal;
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);

  /* Get the rank and coordinates in he new topology */
  MPI_Comm_rank(comm_2d, &my2drank);
  MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

  /* Compute ranks of the up and left shifts */
  MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
  MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

  /* Determine the dimension of the local matrix block */
  nlocal = n/dims[0];

  /* Perform the initial matrix alignment B */
  MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);
  MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);

  /* Get into the main computation loop */
  for (i=0; i<dims[0]; i++) {
    MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/

    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
              leftrank, 1, rightrank, 1, comm_2d, &status);
    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
              uprank, 1, downrank, 1, comm_2d, &status);
  }

  /* Restore the original distribution of a and b */
  MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);
  MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
  MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
         shiftdest, 1, shiftsource, 1, comm_2d, &status);

  MPI_Comm_free(&comm_2d); /* Free up communicator */
}
```

```c
MatrixMultiply(int n, double *a, double *b, double *c) { /* matrix-matrix multiplication c = a*b */
  int i, j, k;
  for (i=0; i<n; i++) for (j=0; j<n; j++) for (k=0; k<n; k++) c[i*n+j] += a[i*n+k]*b[k*n+j];
}
```

# Ο αλγόριθμος του Cannon με μη εμποδιστικές επικοινωνίες

- Θέλει διπλά πινακάκια a και b ώστε σε άλλον buffer να παραλαμβάνει και από άλλον να στέλνει

```
MatrixMatrixMultiply(int n, double *a, double *b, double *c,
                     MPI_Comm comm)
{
 ...
  MPI_Request reqs[4];
  double *a_buffers[2], *b_buffers[2];

  /* Get the communicator related information */
  /* Set up the Cartesian topology */
  /* Set the periods for wraparound connections */
  /* Create the Cartesian topology, with rank reordering */
  /* Get the rank and coordinates in he new topology */
  /* Compute ranks of the up and left shifts */
  /* Determine the dimension of the local matrix block */
  /* Perform the initial matrix alignment B */
  ...

  /* Setup the a_buffers and b_buffers arrays */
  a_buffers[0] = a;
  a_buffers[1] = malloc(nlocal*nlocal*sizeof(double));
  b_buffers[0] = b;
  b_buffers[1] = malloc(nlocal*nlocal*sizeof(double));

  /* Get into the main computation loop */
  for (i=0; i<dims[0]; i++) {
    /* Shift up & left - post early */
    MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
               leftrank, 1, comm_2d, &reqs[0]);
    MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
               uprank, 1, comm_2d, &reqs[1]);
    MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal,
          MPI_DOUBLE, rightrank, 1, comm_2d, &reqs[2]);
    MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal,
          MPI_DOUBLE, downrank, 1, comm_2d, &reqs[3]);

    /* c = c + a*b * /
    MatrixMultiply(nlocal, a_buffers[i%2],
                           b_buffers[i%2], c);

    /* Wait for completion before continuing */
    for (j=0; j<4; j++)
      MPI_Wait(&reqs[j], &status);
  }

  /* Restore the original distribution of a and b */
  ...
}
```