ΥΟ7 – Παράλληλα Συστήματα
2011-12

3/5/2012

Συστήματα κατανεμημένης μνήμης και ο προγραμματισμός τους (II)

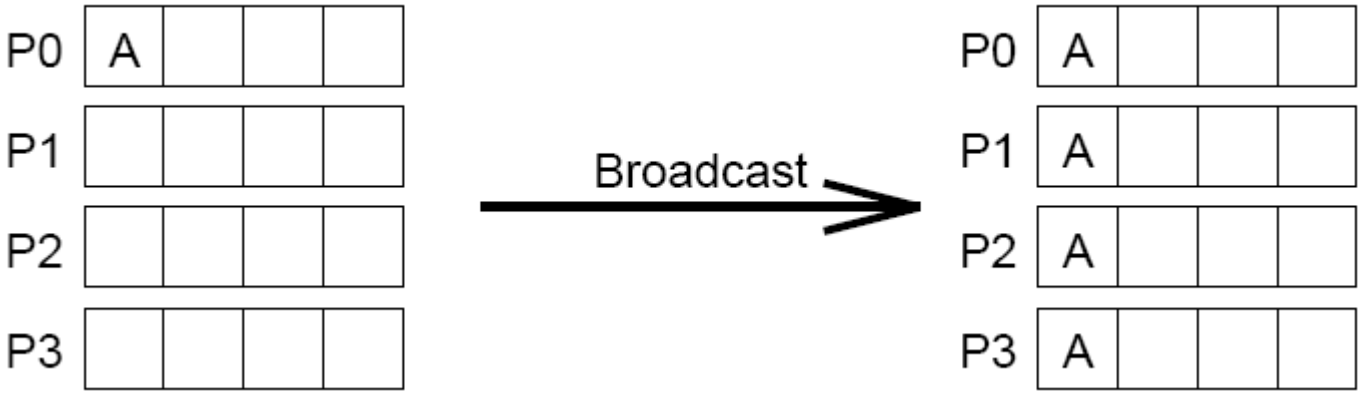
Β. Δημακόπουλος

MPI – συλλογικές επικοινωνίες

Συλλογικές επικοινωνίες (collective communications)

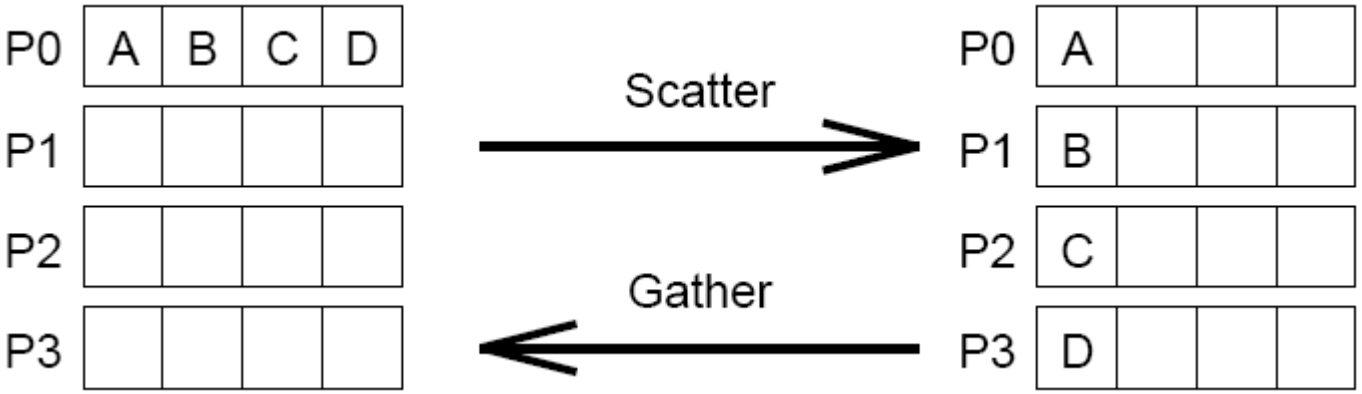
- Εκτός από την επικοινωνία ενός ζεύγους διεργασιών ("unicast communication"), υπάρχει πολύ συχνά ανάγκη για επικοινωνία μεταξύ όλων των διεργασιών μαζί.
  - "collective" communications
  - διάφορων ειδών

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

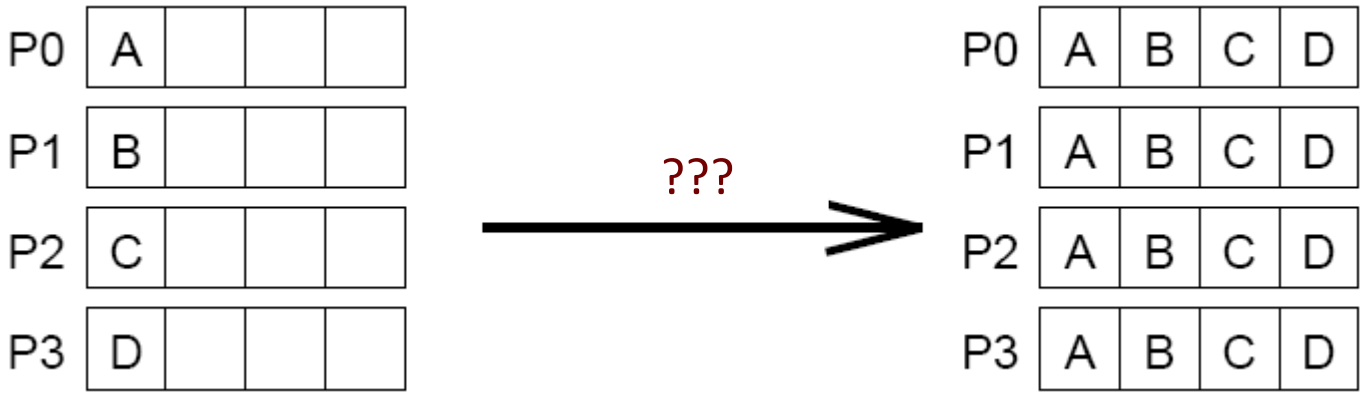


Συλλογικές επικοινωνίες

- Διασκόρπιση (scattering, personalized one-to-all)
  - διαφορετικά μηνύματα από μία διεργασία προς όλες τις άλλες
- Συλλογή (gathering, personalized all-to-one)
  - (αντίθετο της διασκόρπισης)
    μία διεργασία λαμβάνει ένα μήνυμα από κάθε μία από τις υπόλοιπες

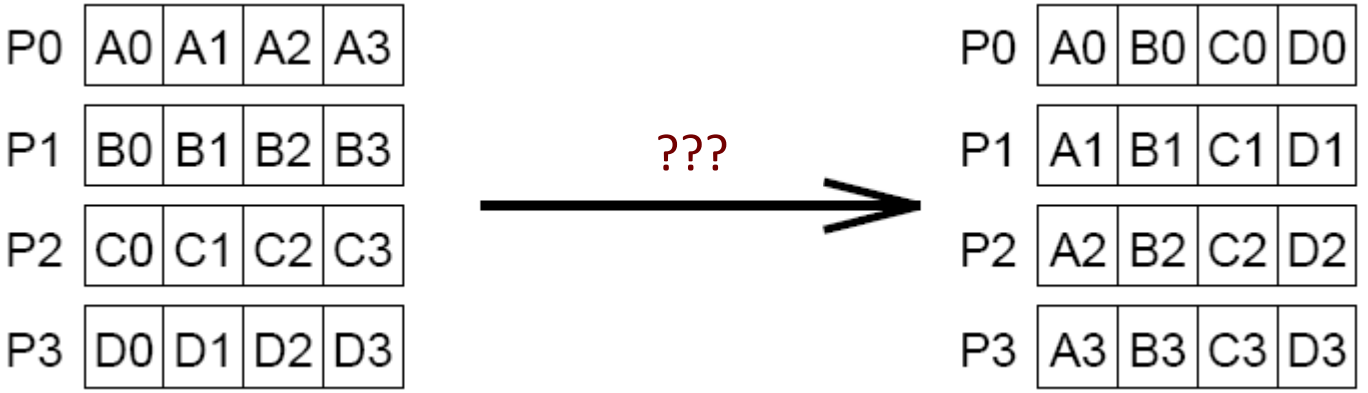


Συλλογικές επικοινωνίες



- 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**

Παράδειγμα

- Σε πολλές εφαρμογές απαιτείται η λύση ενός συστήματος εξισώσεων και η συνήθης τακτική για τη λύση του είναι οι λεγόμενες «επαναληπτικές μέθοδοι». Αυτέ πολύ απλά προσεγγίζουν σταδιακά τη λύση x(), μέσω επαναλήψεων της μορφής:

$$x(t+1) = A \cdot x(t)$$

  ξεκινώντας από μία αρχική εκτίμηση x(0).

- Το x() είναι διάνυσμα με N στοιχεία και το A είναι πίνακας NxN. Άρα σε κάθε επανάληψη απαιτείται πολλαπλασιασμός πίνακα επί διάνυσμα.

Παράδειγμα, συνέχεια

- Μία στρατηγική αποθήκευσης των A και x():
  - Κατά γραμμές ο A (π.χ. η διεργασία i έχει τη γραμμή i του A)
  - Κάθε διεργασία έχει όλο το τρέχον x()

- Αρχικά, κάποια διεργασία πρέπει:
  - να στείλει σε όλες τις άλλες την αρχική εκτίμηση (το x(0)).
    (*εκπομπή*)
  - να στείλει στην διεργασία i την i-οστή γραμμή του A (για κάθε i).
    (*διασκόρπιση*)

Παράδειγμα, συνέχεια

- Σε κάθε επανάληψη:
  - Η διεργασία i πολλαπλασιάζει τη γραμμή του A που έχει με το x(t) που επίσης έχει και υπολογίζει το i-οστό στοιχείο του νέου x, δηλαδή το $x_i(t+1)$.
  - Θα πρέπει να σχηματίσει ολόκληρο το x(t), προφανώς από τα στοιχεία που υπολόγισαν οι άλλες διεργασίες

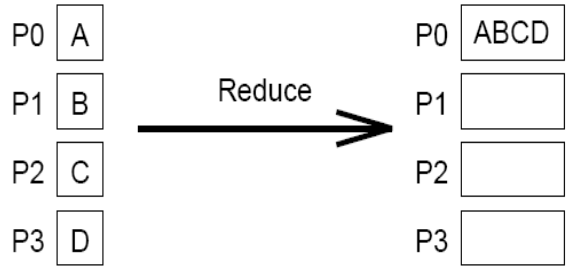
    (*πολλαπλή εκπομπή ή allgather*)

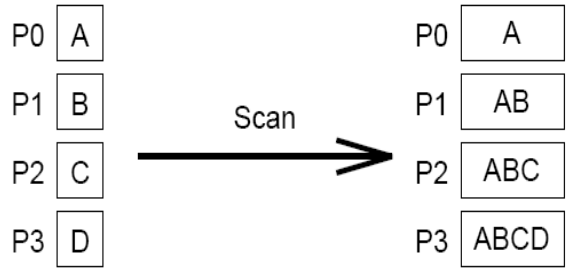
- Αν απαιτηθεί *αναστροφή* του A (οι γραμμές να γίνουν στήλες και οι στήλες γραμμές), τότε η διεργασία i θα έχει μόνο το στοιχείο A_{ii}.
  - Το A_{0i} το έχει η διεργασία 0, το A_{1i} το έχει η διεργασία 1, το A_{ki} το έχει η διεργασία k.
  - Πρέπει να *συλλέξει* αυτά τα στοιχεία από όλες τις άλλες διεργασίες
  - Το ίδιο κάνουν, όμως, όλες, για τις δικές τους στήλες

    (*ολική ανταλλαγή ή alltoall*)

Άλλες συλλογικές επικοινωνίες/υπολογισμοί

- Λειτουργίες υποβίβασης (reduction operations)
  - Από ένα 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**).

- `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

Κλήσεις, συνέχεια

- `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: «κυριλέ» (☺) υπολογισμός του π

```
#include <mpi.h>   /* Μετάφραση με mpicc */

main(int argc, char *argv[])
{
  float      W, result = 0.0, pi;
  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);

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

  /* Ο υπολογισμός της κάθε διεργασίας */
  W = 1.0 / N;
  for (i = myid; i < N; i += nproc)
    result += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);

  /* Συλλογή αποτελεσμάτων */
  MPI_Reduce(&result, &pi, 1, MPI_FLOAT,
             MPI_SUM, 0, MPI_COMM_WORLD);

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

  MPI_Finalize();
}
```

Παράδειγμα: πίνακας επί διάνυσμα (II) – συλλογικά

```
matrix_times_vector()
{
  int        i, j;
  int        WORK = N / nprocs;      /* Στοιχεία ανά διεργασία */
  float      sum, v[N], A[N][N], **B;

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

  B = allocmatrix(WORK, N);
  mypart = allocvector(WORK);
  MPI_Scatter(A, WORK*N, MPI_FLOAT, B, WORK*N, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(v, N, MPI_FLOAT, 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_FLOAT, res, WORK, MPI_FLOAT, 0, MPI_COMM_WORLD);

  if (myid == 0)
    show_result(res);
}
```

MPI – «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 το ποσό των διαθέσιμων buffers
    - ΜΗ ΑΣΦΑΛΕΣ ΠΡΟΓΡΑΜΜΑ

- Παρόμοια περίπτωση είναι και όταν Ν 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 – Μονόπλευρες επικοινωνίες
(one-sided communications)

One-sided communications

- Μέχρι τώρα *αμφίπλευρες* επικοινωνίες (εμπλέκεται και ο αποστολέας – send – και ο παραλήπτης – receive).
- Στις μονόπλευρες εμπλέκεται μόνο ο ένας από τους δύο
  - One-sided communications ή
  - Remote memory access (RMA)

- MPI_Put()
- MPI_Get()

- Λεπτομέρειες;
  - Homework!

MPI – προχωρημένες λειτουργίες

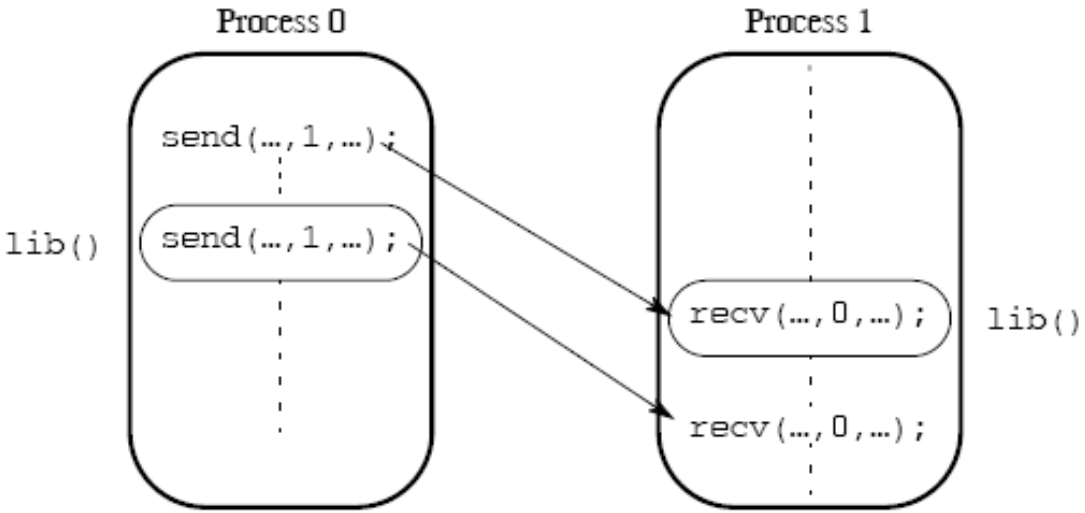
Communicators

- Αν θέλω να ορίσω γκρουπ 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 αποστολές 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.

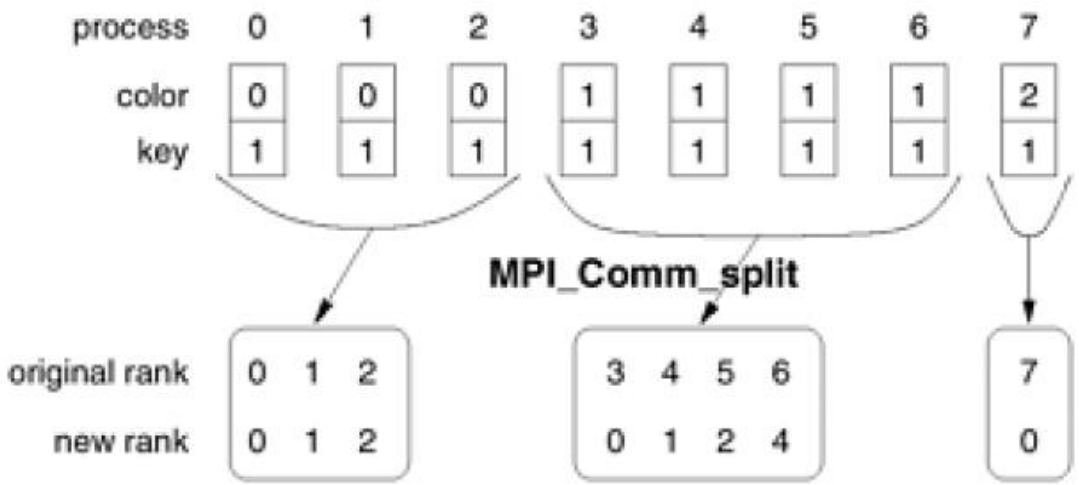
Φτιάχνοντας communicators

- 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:

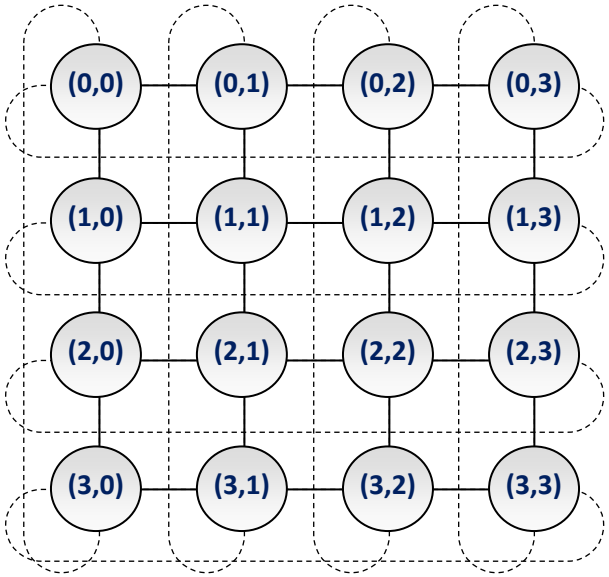
```
MPI_Cart_create(MPI_Comm comm_old, int ndims,
    int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
```

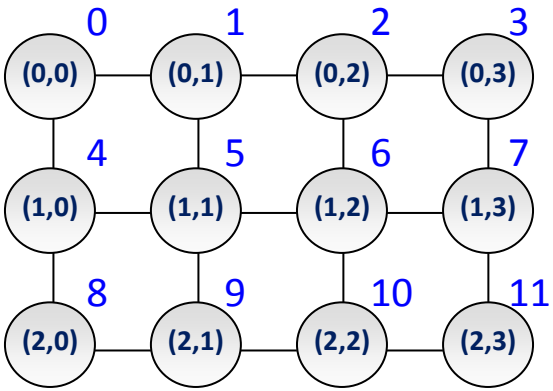
Οι διεργασίες που ανήκουν στον παλιό communicator δημιουργούν ένα νέο communicator με τοπολογία πλέγματος “ndims” διαστάσεων.

  - 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:

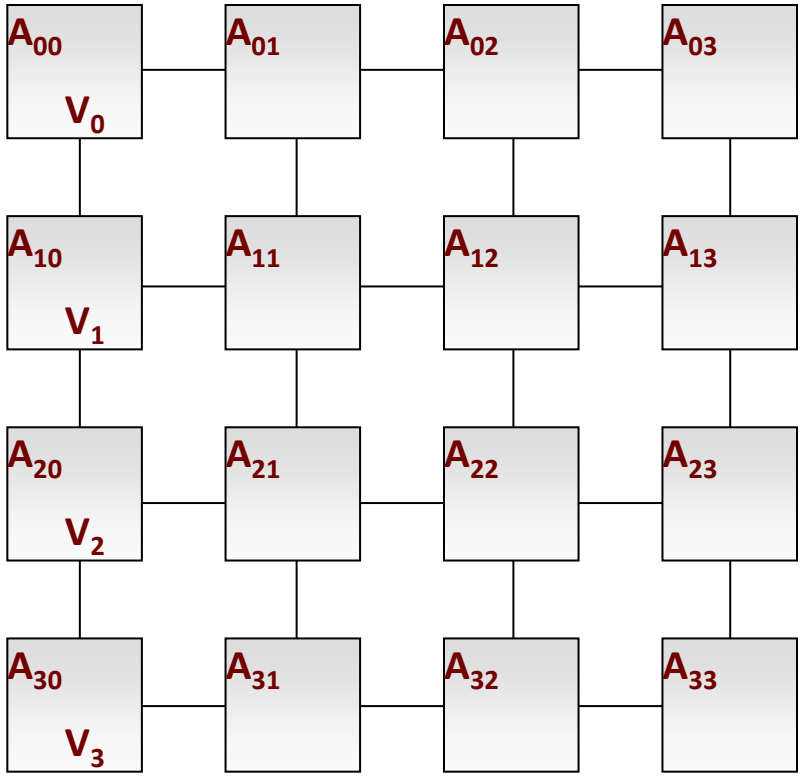
```
MPI_Cart_shift(MPI_Comm cartcomm, int dir, int s_step,
               int *rank_source, int *rank_dest)
```

  - 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().*

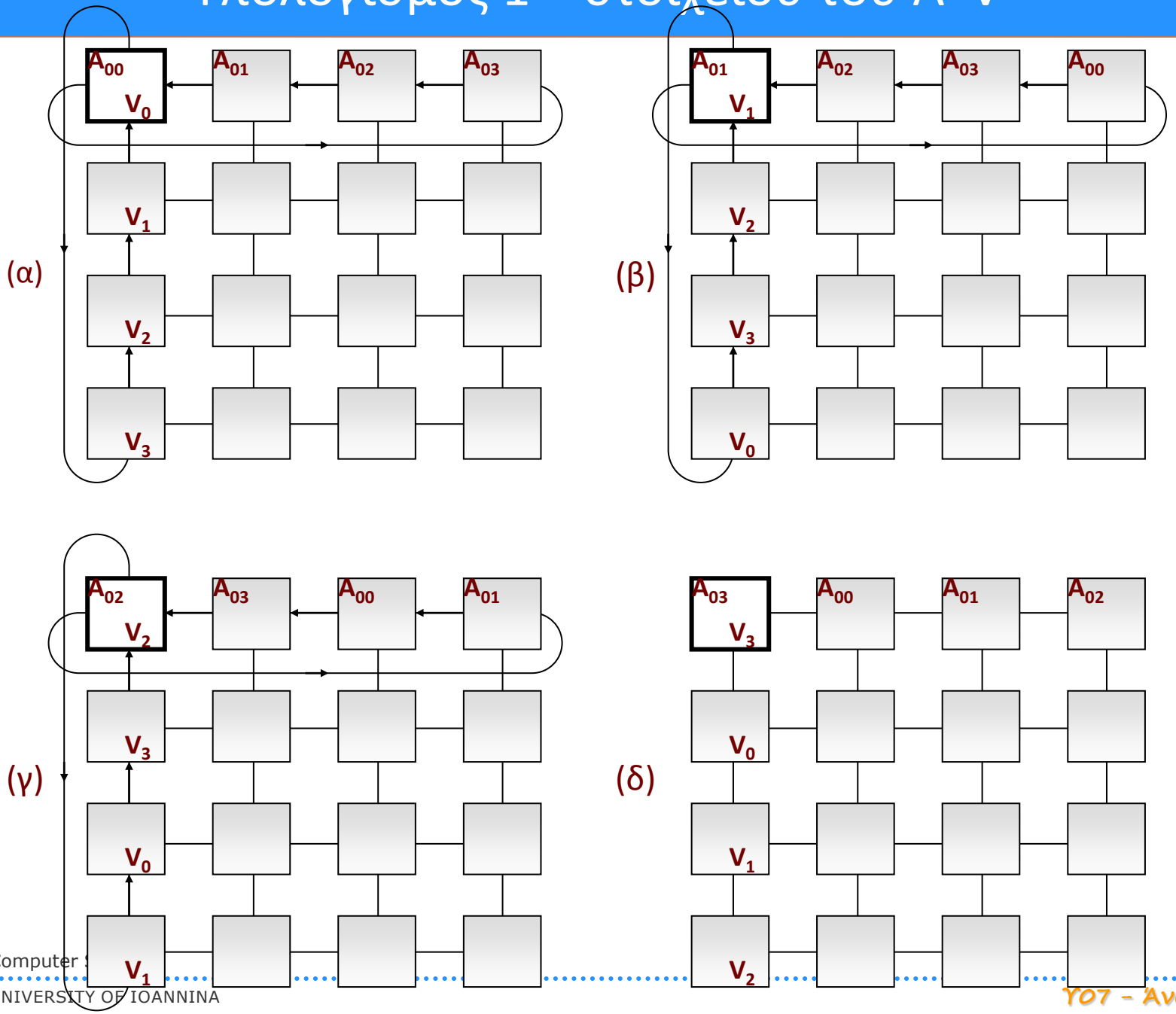
Εφαρμογή: ο αλγόριθμος του Cannon για πολλαπλασιασμό πινάκων

Πολλαπλασιασμός A*v

- Αρχική τοποθέτηση στοιχείων (πλέγμα/torus)



Υπολογισμός 1ου στοιχείου του A*v



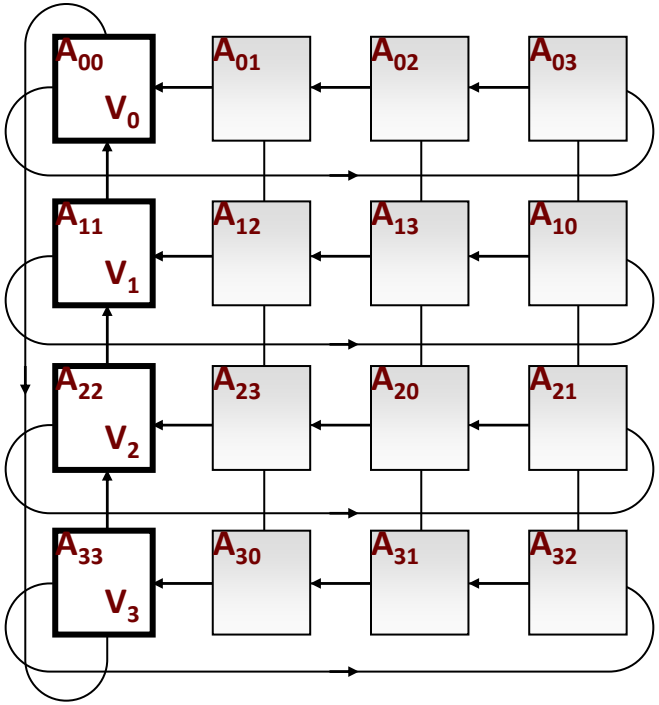
Τι λείπει για να υπολογιστεί το 2ο στοιχείο;

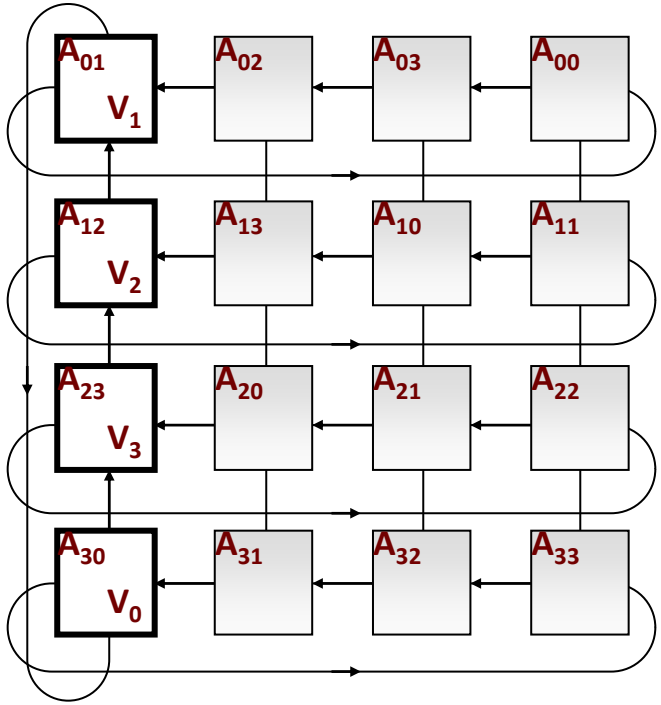
- Να γίνει το αντίστοιχο με την γραμμή 1 του A
- Όμως πρέπει να ξεκινήσουμε από το A[1][1]*v[1], μιας και το v[1] είναι αυτό που υπάρχει στην γραμμή 1. Πώς;
  - Απάντηση: *πριν* ξεκινήσει ο αλγόριθμος, κάνουμε τη γραμμή 1 SHIFT (rotate) 1 θέση προς τα αριστερά. Από εκεί και ύστερα, ο υπολογισμός του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

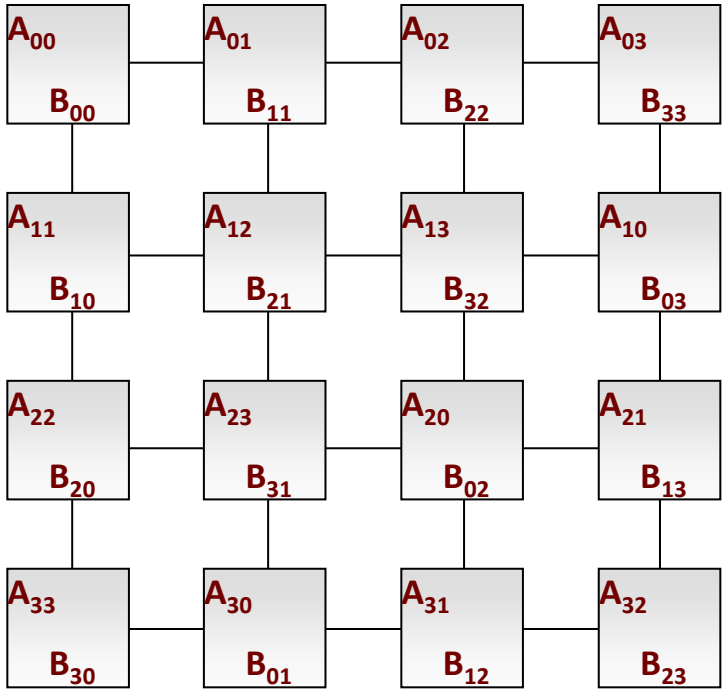
A*v – αρχικές τοποθετήσεις & 1ο βήμα





Γενίκευση σε A*B (αλγόριθμος Cannon)

- Αρχικά,
  - κάθε γραμμή *i* του πίνακα Α ολισθαίνει κατά *i* θέσεις *αριστερά* και
  - κάθε στήλη *j* του πίνακα Β 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

```
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 */
}
```

```
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() «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 από τον χώρο του χρήστη (user space), το τοποθετεί πιθανώς σε δικούς της buffers
    - Το λειτουργικό σύστημα αντιγράφει από τον χώρο της βιβλιοθήκης σε buffers στον χώρο του πυρήνα (kernel space)
    - Από τους buffers της μνήμης του πυρήνα 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, κάποια από τις μεταφορές στους διάφορους buffers του τοπικού κόμβου ίσως να μην έχει ολοκληρωθεί!
  - *Μία νέα MPI_Isend() «στο καπάκι» μπορεί να καταστρέψει το προηγούμενο μήνυμα που δεν έχει προλάβει ακόμα να φύγει*
  - Πρέπει να σιγουρευτούμε ότι το προηγούμενο μήνυμα έχει φύγει πριν στείλουμε το επόμενο:

```
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 λόγω 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!

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

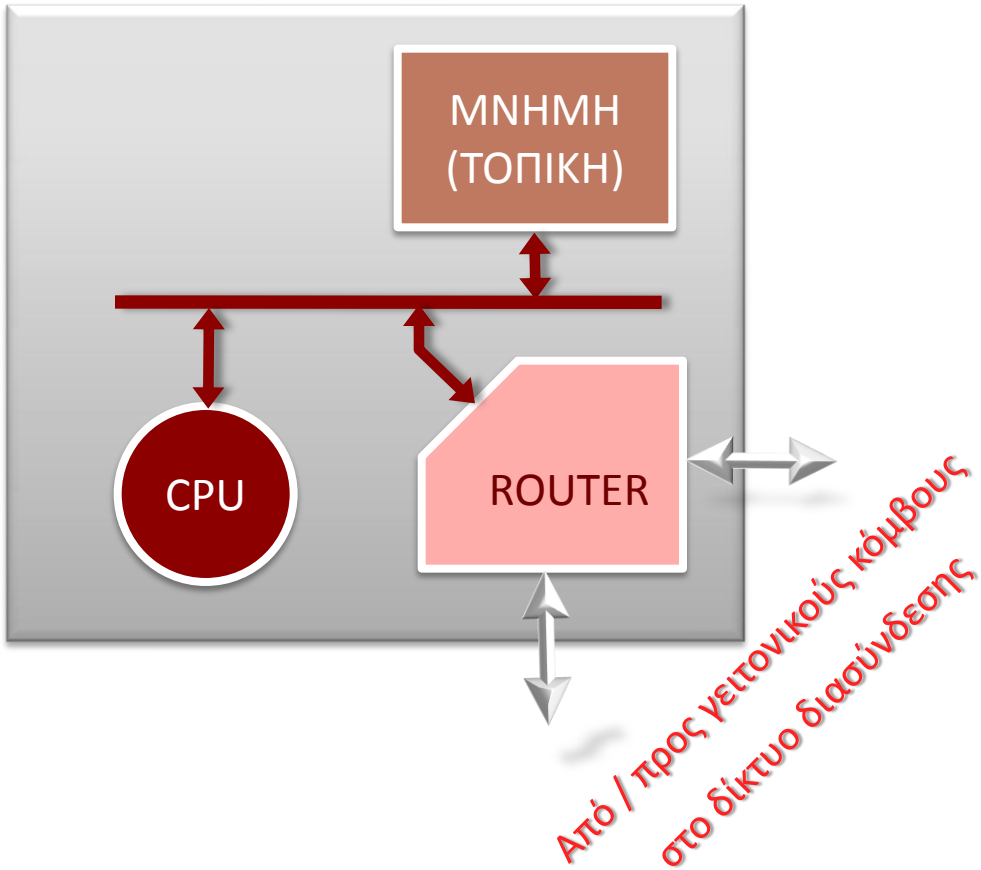
```
MatrixMatrixMultiply(int n, double *a, double *b, double *c,
                     MPI_Comm comm)
{
  ...
  MPI_Request reqs[4];
  double *a_buffers[2], *b_buffers[2];

  /* 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 */
  ...
}
```

Distributed shared memory (DSM)
ή
Shared Virtual Memory (SVM)

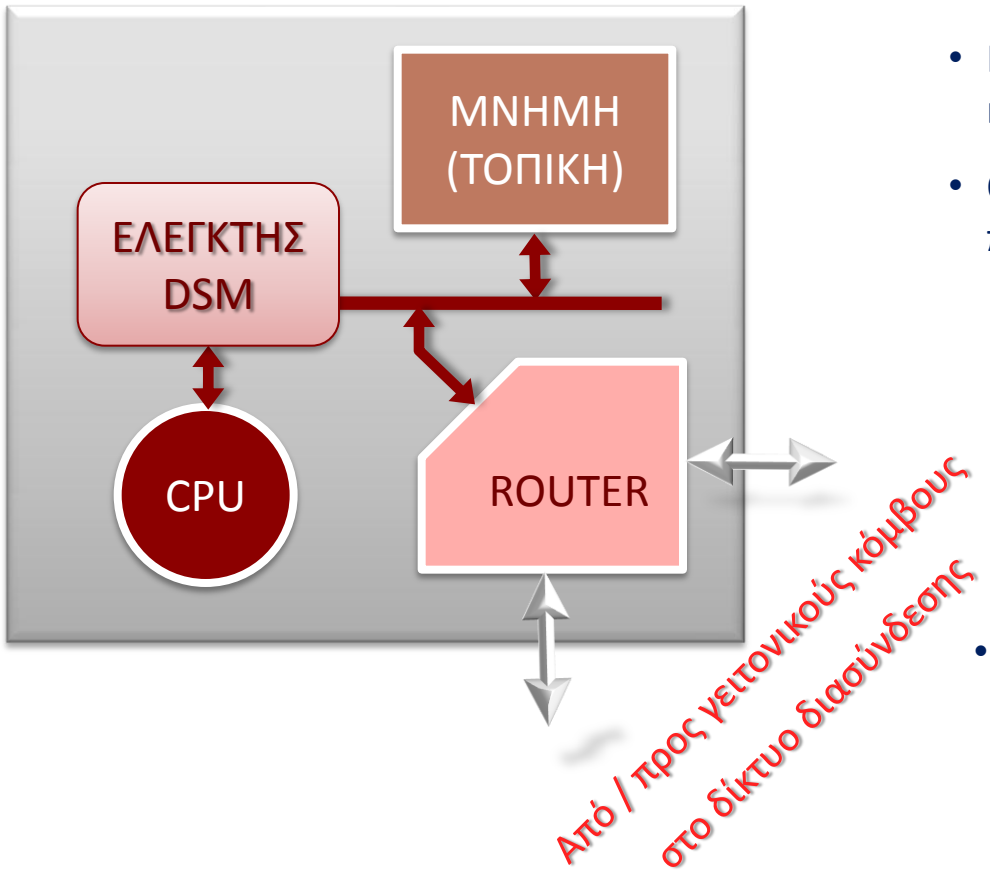
Γιατί;

- Συστήματα κατανεμημένης μνήμης:
  - Αρχιτεκτονική: **κλιμακώσιμη**
  - Προγραμματισμός (MPI): αρκετά **δύσκολος** αλλά και δυνατότητα επιδόσεων
- Συστήματα κοινής μνήμης (SMPs):
  - Αρχιτεκτονική: **δύσκολα κλιμακώσιμη**
  - Προγραμματισμός: **σχετικά εύκολος**
- Το ιδεώδες:
  - Κλιμακώσιμες αρχιτεκτονικές που να προγραμματίζονται εύκολα (αλλά μην ξεχνάμε και τις επιδόσεις)
  - Μετά το σειριακό, το πιο εύκολο είναι ο προγραμματισμός κοινής μνήμης (π.χ. OpenMP)
- Λύση: *«εξομοίωση» κοινής μνήμης πάνω από το σύνολο των ιδιωτικών μνημών*
  - με hardware
  - με software

Hardware – κόμβος συστήματος



Hardware – ελεγκτής DSM



- Η CPU προσπελαύνει όλο τον χώρο διευθύνσεων ενιαία
- Η τοπική μνήμη έχει μόνο ένα μικρό κομμάτι του χώρου
- Ο ελεγκτής DSM ελέγχει κάθε διεύθυνση που προσπελαύνει η CPU
  - (1) Αν είναι για την τοπική μνήμη δεν κάνει τίποτε
  - (2) Αν όχι, αναλαμβάνει την επικοινωνία με τον κόμβο που την χειρίζεται, στέλνοντας κατάλληλο μήνυμα. Μόλις έρθει η απάντηση, δίνει δεδομένα στη CPU σαν να ήταν αποθηκευμένο τοπικά
- Το μόνο που καταλαβαίνει η CPU είναι η διαφορά στην ταχύτητα προσπέλασης κάποιων δεδομένων (τα απομακρυσμένα κάνουν πολύ παραπάνω χρόνο να έρθουν)
  - NUMA (non-uniform memory access)
- Χρήση cache στον ελεγκτή για τα απομακρυσμένα δεδομένα
  - ccNUMA (cache coherent NUMA)

Software DSM

- Σε clusters?
  - Δεν υπάρχει δυνατότητα επέμβασης στο hardware
  - Μόνο λύσεις software
- Κλασική υλοποίηση: *page-based sDSM*
  - Χωρίς παρεμβάσεις στο Λ.Σ. (είναι μία απλή εφαρμογή σε επίπεδο χρήστη)
  - Όλοι οι συμμετέχοντες κόμβοι «δεσμεύουν» έναν χώρο μνήμης (πολλές *σελίδες*) που θα τον έχουν ως κοινόχρηστο
    - Τα κοινόχρηστα δεδομένα θα τοποθετούνται σε αυτό τον χώρο
  - Καθένας «μαρκάρει» τις σελίδες ως «απαγορευμένης προσπέλασης» και ορίζει έναν χειριστή σημάτων για το σήμα SIGSEGV
  - Η εφαρμογή ξεκινά και οι προσπελάσεις στον κοινόχρηστο χώρο οδηγούν σε σφάλματα μνήμης που εκκινούν τον χειριστή για το SIGSEGV.
  - Ο χειριστής:
    - επικοινωνεί με τον κόμβο που διαθέτει το δεδομένο, φέρνοντας ολόκληρη τη σελίδα που το περιέχει
    - μετά τη λήψη και την τοποθέτηση στη σωστή θέση, αλλάζει τα δικαιώματα της σελίδας (π.χ. επιτρέπεται πλέον η ανάγνωση)
    - τελειώνει, επανεκτελώντας την εντολή που προκάλεσε τη διακοπή

Ιδέες για υλοποιήσεις page-based sDSM

- Μία σελίδα p θα υπάρχει μόνο σε έναν κόμβο
  - Κεντρικός server που γνωρίζει ποιος κόμβος έχει ποια σελίδα
    - Απορρίπτεται – μεγάλη κίνηση στον server
  - Κατανεμημένα & προκαθορισμένα – π.χ. ο κόμβος p % N είναι υπεύθυνος για γνωρίζει που βρίσκεται η σελίδα p
    - Καλύτερο, μιας και κάθε κόμβος μπορεί να ρωτήσει κατευθείαν αυτόν που πρέπει
  - Αν υπάρχουν πολλοί readers, σε κάθε προσπέλαση «σελίδες πάνε κι έρχονται»
- Λύση: πολλαπλά αντίγραφα σελίδων (page replication)
  - Απλούστερη υλοποίηση: multi-readers **Ή** 1single-writer
    - Αν εμφανιστεί writer, τότε όλα τα read-only copies ακυρώνονται
    - Στο write υπάρχει *μεγάλο* overhead
    - Πολύ καλό αν υπάρχουν πολλά reads και ελάχιστα writes
  - Καλύτερη (σε γενικές γραμμές) υλοποίηση: multi-readers **ΚΑΙ** multi-writers
    - Πιο δύσκολος ο χειρισμός των πολλαπλών εγγραφέων
  - Conherency & consistency problems!!!