ΥΟ7 – Παράλληλα Συστήματα
2011-12

30/4/2012

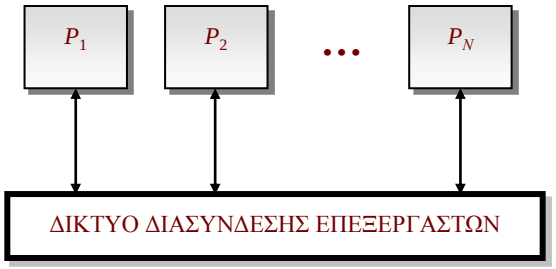
Συστήματα κατανεμημένης μνήμης και ο προγραμματισμός τους (Ι)

Β. Δημακόπουλος

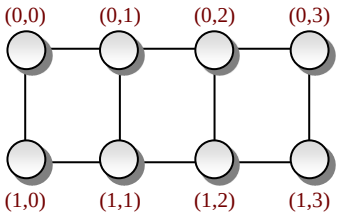
multicomputers, MPPs, clusters

Πολυεπεξεργαστές κατανεμημένης μνήμης

- Ανεξάρτητοι επεξεργαστές, ο καθένας με την ιδιωτική του μνήμη (κόμβος = CPU + μνήμη)



- Δίκτυο διασύνδεσης επεξεργαστών (interconnection network)
  - δίαυλος
  - δίκτυο διακοπτών
  - point-to-point, στατικό, άμεσο δίκτυο



Βασική οργάνωση

- *Επικοινωνία επεξεργαστών μέσω ανταλλαγής μηνυμάτων, επάνω από το δίκτυο διασύνδεσης*

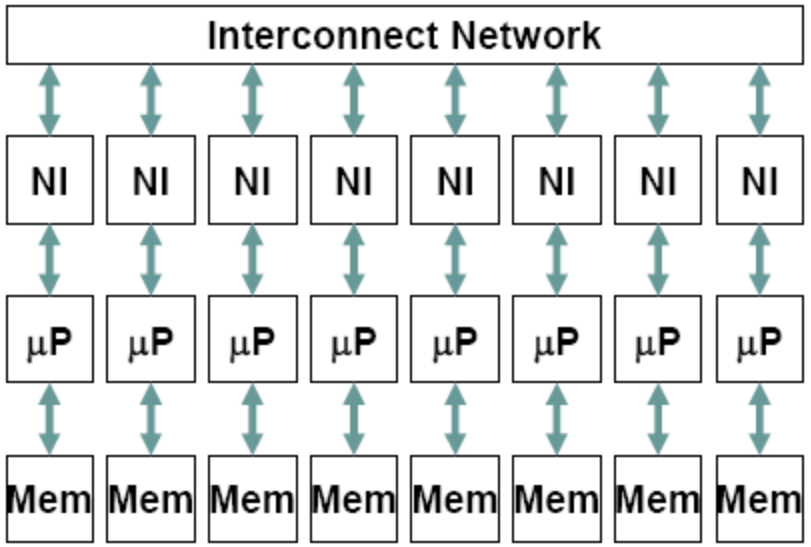
- Λόγω του ότι κάθε κόμβος είναι ουσιαστικά ένας (σχεδόν) ολοκληρωμένος και αυτόνομος υπολογιστής, οι ΠΚΜ είναι γνωστοί και ως *πολυϋπολογιστές* (multicomputers)

- Η οργάνωση μοιάζει με δίκτυο υπολογιστών
  - Διαφορές:
    - ταχύτητα
    - τοπολογία
    - λειτουργικό σύστημα
    - ...

Massively Parallel Processors (MPPs)

- Initial Research Projects
  - Caltech Cosmic Cube (early 1980s) using custom Mosaic processors
- Commercial Microprocessors including MPP Support
  - Transputer (1985)
  - nCube-1(1986) /nCube-2 (1990)
- Standard Microprocessors + Network Interfaces
  - Intel Paragon (i860)
  - TMC CM-5 (SPARC)
  - Meiko CS-2 (SPARC)
  - IBM SP-2 (RS/6000)
- MPP Vector Supers
  - Fujitsu VPP series

Designs scale to 100s or 1000s of nodes

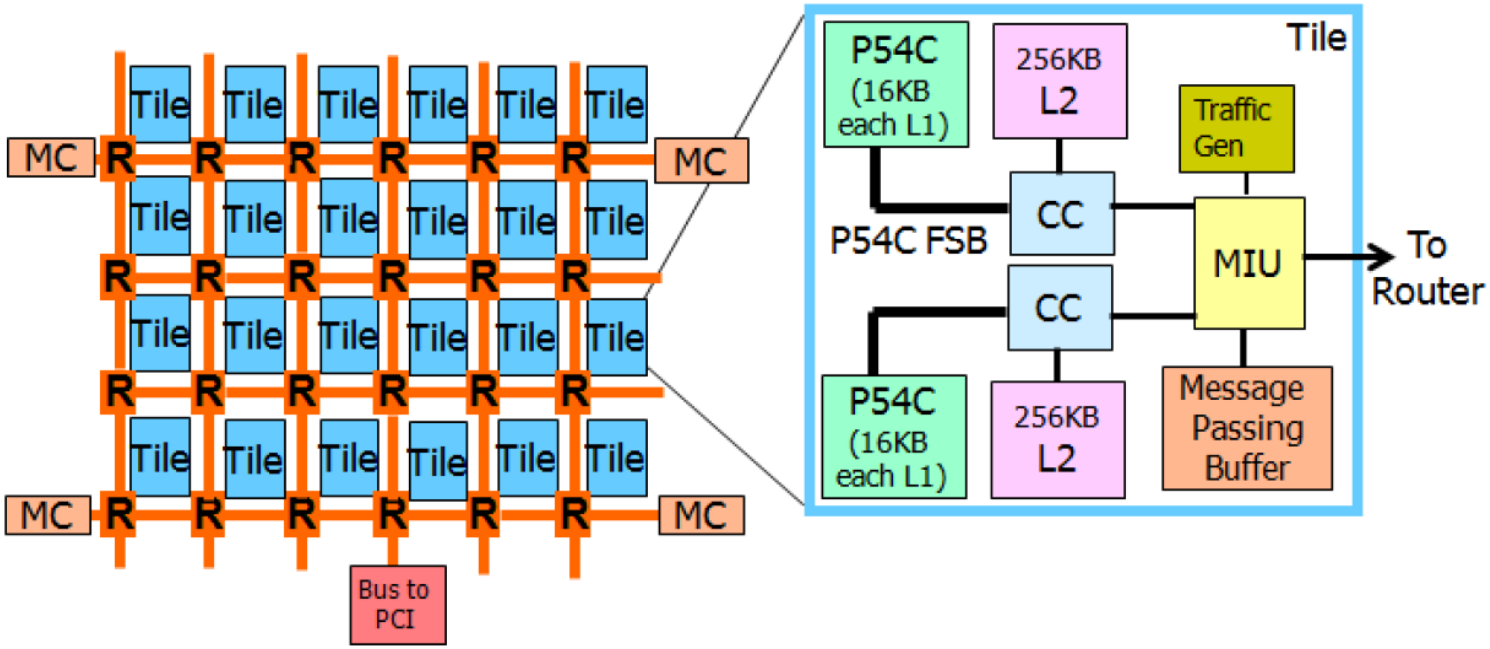


Clusters

- Παντού!
  - Συλλογή από διασυνδεδεμένους «κόμβους»
    - Φτηνοί / ευρέως διαθέσιμοι επεξεργαστές (π.χ. Clusters από PCs)
  - Ο μόνος τρόπος να φτιάξουμε «οικονομικούς» υπερ-υπολογιστές (πολλά Teraflops)
  - Πολύ λίγοι έως πάρα πολλοί κόμβοι
  - Sandia Laboratories Red Storm (Cray, 2004)
    - 13000 AMD Opterons (basically PC nodes), 75 Terabytes of memory
    - > 100 Teraflops (peak)
    - Linux
    - Κόστος: $90.000.000
- Στο τμήμα μας:
  - Γενικότερο δίκτυο σταθμών εργασίας (100Mbps ethernet, αργό με πολύ κίνηση)
  - Αυτόνομο cluster (http://gatepc73.cs.uoi.gr:8880)
    - 16 κόμβοι, κάθε κόμβος 2 CPUS, κάθε CPU διπύρηνη
    - gigabit Ethernet

Manycore

- Πάρα πολλοί πυρήνες
  - Π.χ. Intel TeraScale I (80-cores),
    TeraScale II (SCC, 48-cores / 24 tiles– see below)
  - Mesh network (NoC)



το δίκτυο διασύνδεσης

Τι θέλουμε από έναν δίκτυο διασύνδεσης …

- Το δίκτυο διασύνδεσης θα πρέπει να μεταφέρει *όσο το δυνατόν περισσότερα* μηνύματα, *όσο το δυνατόν γρηγορότερα* με *ελάχιστο κόστος* και *μέγιστη αξιοπιστία*. Αυτά είναι αλληλοσυγκρουόμενα, όμως.

Τοπολογία:

- Μικρή διάμετρος, μικρή μέση απόσταση

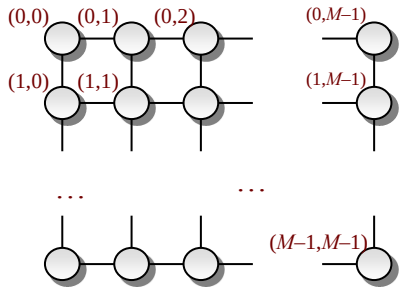
  μικρή καθυστέρηση σε packet-switching, μικρή contention σε wormhole switching

- Μικρός και σταθερός βαθμός

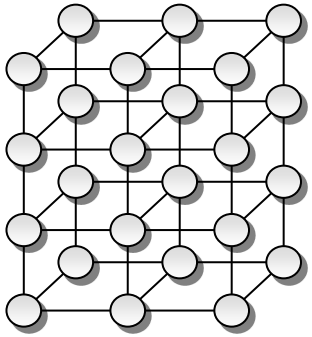
  απλοί και οικονομικοί routers, μικρότερη και σταθερή καλωδίωση, χαμηλότερη connectivity, μεγαλύτερες αποστάσεις

- Μεγάλο bisection width
- Υψηλό connectivity
- Συμμετρία
- Εύκολη ενσωμάτωση άλλων γράφων και σε άλλους γράφους

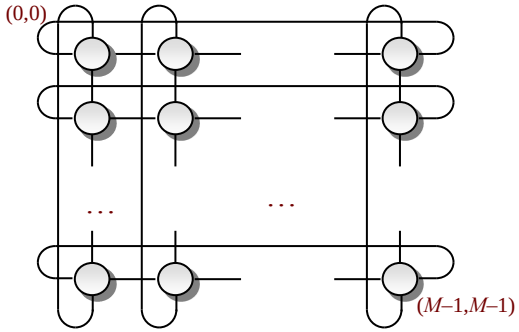
Συνήθεις τοπολογίες στην πράξη: πλέγματα, tori, κύβοι



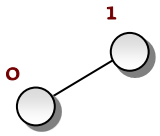
Πλέγμα M×M



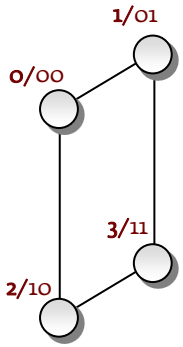
Πλέγμα 4×3×2



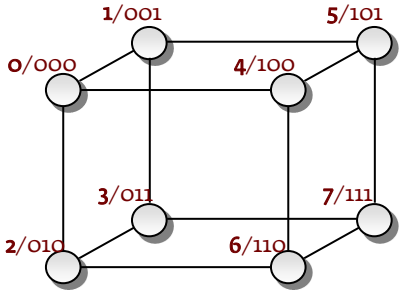
Torus M×M



Μονοδιάστατος κύβος (Q_1)



Δισδιάστατος κύβος (Q_2)



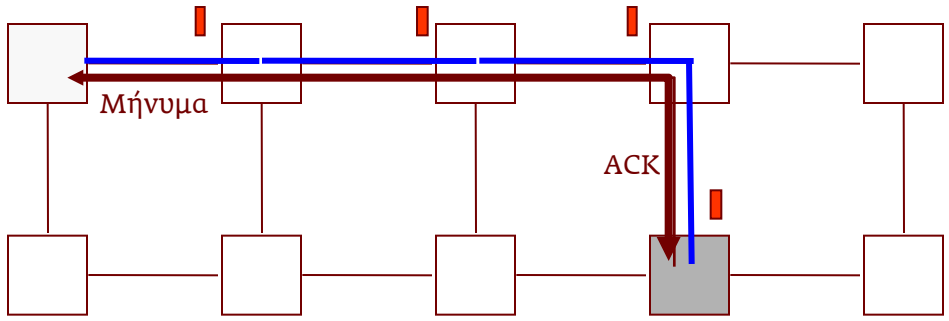
Τρισδιάστατος κύβος

Ένα δίκτυο διασύνδεσης χαρακτηρίζεται από:

- Την *τοπολογία* του
  - Ποίος κόμβος συνδέεται με ποιον
- Τη *διαδρόμησή* του (routing)
  - Ποιο από όλα τα δυνατά μονοπάτια θα επιλεχθεί
    - Πολλές επιλογές πολιτικών
- Τον *έλεγχο ροής* του (flow control)
  - Πώς διανέμονται οι πόροι του δικτύου (κανάλια, buffers κλπ), τι συμβαίνει σε περίπτωση συγκρούσεων
    - Αρχιτεκτονική του διαδρομητή
- Τη *μεταγωγή* του (switching)
  - Πώς μεταφέρεται εσωτερικά σε έναν διαδρομητή το μήνυμα από μία είσοδο σε μία έξοδό του
    - Κυκλώματος (circuit switching)
    - Πακέτου / μηνύματος / SAF (Store-and-Forward)
    - Virtual Cut-Through (VCT)
    - Wormhole
    - Virtual channels
    - Pipelined circuit switching
    - …

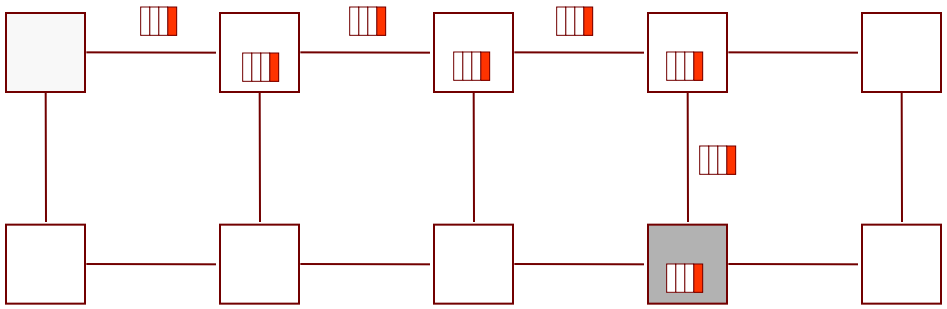
Μεταγωγή κυκλώματος

- Τρεις φάσεις:
  - σχηματισμός (και δέσμευση) του μονοπατιού από το probe
  - μεταφορά του μηνύματος
  - αποδέσμευση του μονοπατιού



Μεταγωγή SAF

- Πακέτου / μηνύματος / SAF (Store-and-Forward)
  - Το μήνυμα χωρίζεται σε πακέτα σταθερού μήκους
  - Κάθε πακέτο προωθείται ανεξάρτητα.
  - Οι κόμβοι
    - (α) το λαμβάνουν και το αποθηκεύουν σε buffer και
    - (β) το προωθούν στον επόμενο κόμβο



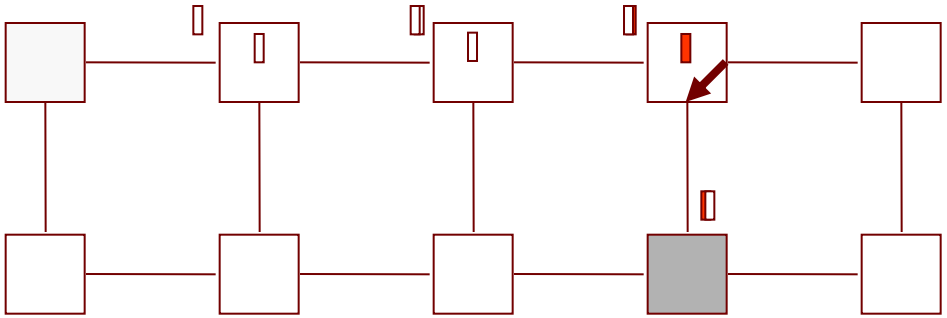
Μεταγωγή VCT

- Σαν το SAF αλλά:
  - Αν το κανάλι εξόδου είναι ελεύθερο, καθώς λαμβάνονται τα bits της επικεφαλίδας, αποφασίζεται το κανάλι εξόδου και όλο το μήνυμα διοχετεύεται κατευθείαν εκεί (άρα ελάχιστη καθυστέρηση).
  - Αν όχι, buffering όπως στο SAF.

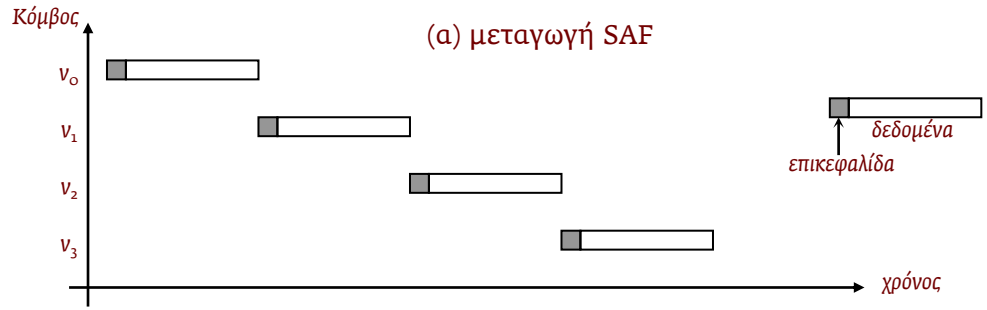
  - Ταχύτητα αν δεν υπάρξει εμπόδιο
  - Όμως, δεν εξαλείφεται η ανάγκη για buffers που έχει και το SAF.

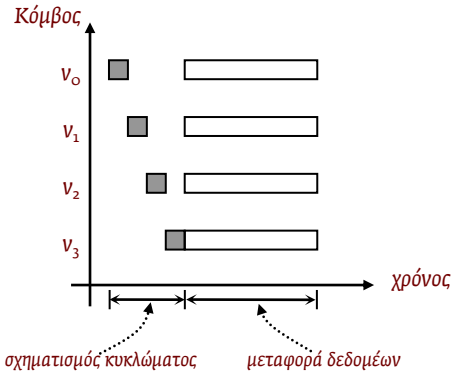
Μεταγωγή wormhole

- Ανάμεσα σε VCT και circuit switching. Το μήνυμα χωρίζεται σε ΠΟΛΥ μικρά πακέτα, τα *flits* (1-4 bytes). Το πρώτο αποτελεί την επικεφαλίδα
- Η επικεφαλίδα προχωρά με VCT αλλά τα υπόλοιπα flits ακολουθούν (και δεσμεύουν) τους προηγούμενους κόμβους, χωρίς κενά, σαν σε pipeline.
- Αν η επικεφαλίδα μπλοκάρει κάπου, τα flits αποθηκεύονται *εκεί που βρίσκονται* (άρα πολύ μικροί buffers απαιτούνται).

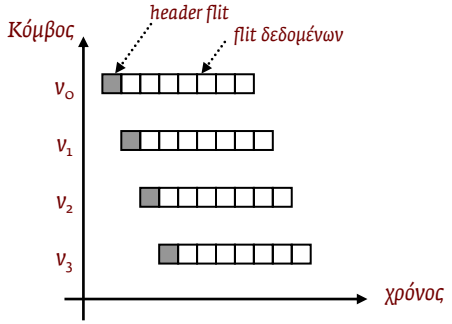


Χρονισμός





(β) μεταγωγή κυκλώματος



(γ) μεταγωγή wormhole

Σύγχρονα δίκτυα υψηλών ταχυτήτων

- Το wormhole switching έχει επικρατήσει διότι
  - Ταχύτητα ακόμα και σε δίκτυα μεγάλης διαμέτρου
  - Ελάχιστο buffering
  - Οπότε γίνεται δυνατή η *υλοποίηση routers σε ανεξάρτητο chip και όχι μέσω της μνήμης του κόμβου*
    - *High-speed (low latency) routers and networks*

- Δίκτυα χαμηλής καθυστέρησης
  - Π.χ. Myrinet, Infiniband κλπ.
  - Πολύ ακριβότερα

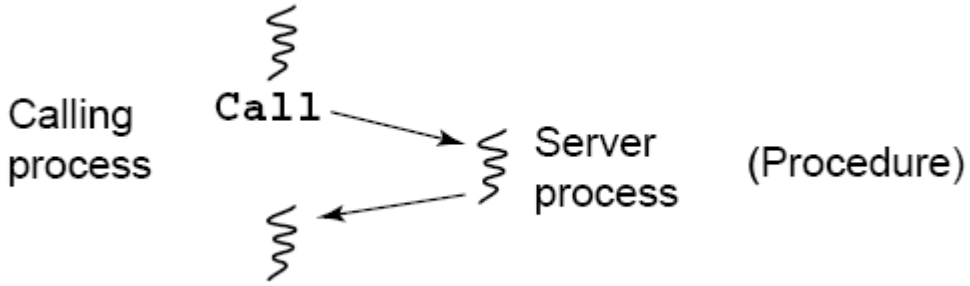
Προγραμματίζοντας …

«Κατηγορίες» προγραμματισμού κατανεμημένης μνήμης

- 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”.
- Ο 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
- Ο 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

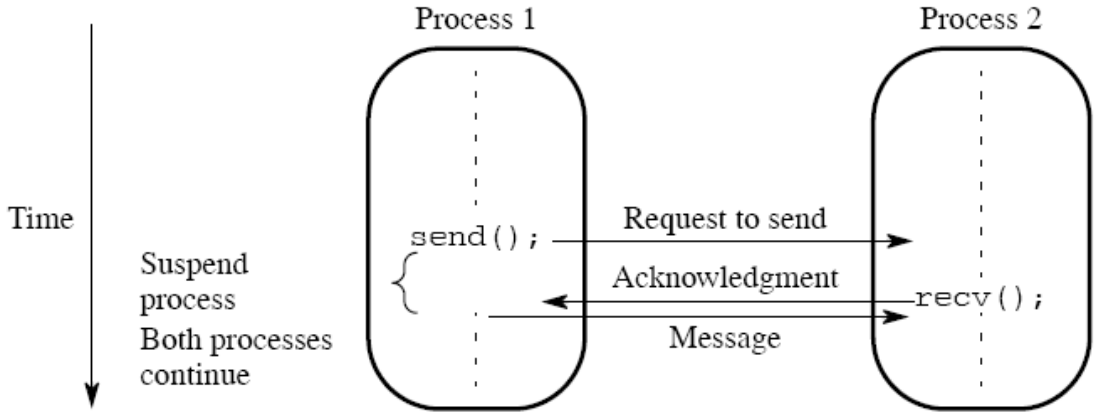
Μεταβίβαση μηνυμάτων: μόνο 2 συναρτήσεις είναι αρκετές …

- Αποστολή μηνύματος – send()
- Παραλαβή μηνύματος – receive()

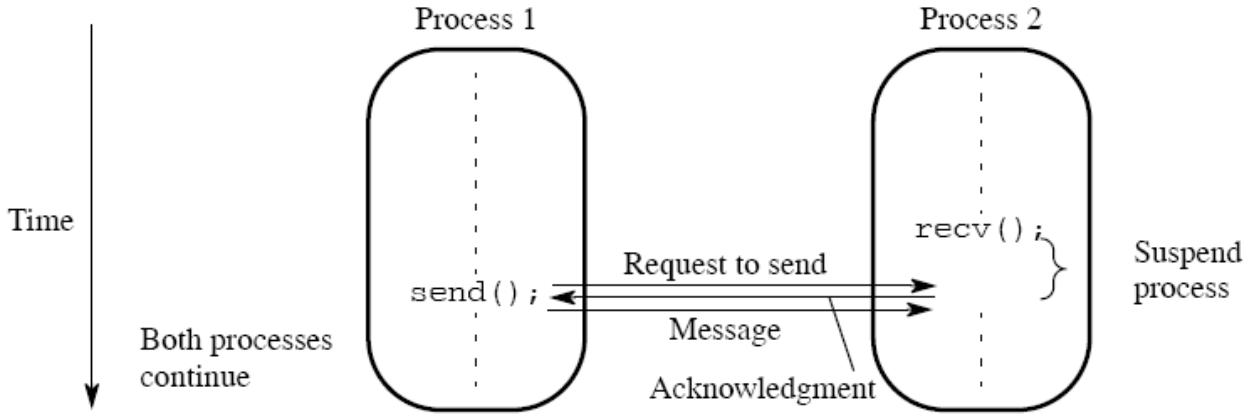
- Θέματα που μπαίνουν:
  - Ταχύτητα επικοινωνιών
    - Εξαρτάται από το δίκτυο και τα χαρακτηριστικά του
    - Εξαρτάται και από τη βιβλιοθήκη που υλοποιεί τις send() και receive()
      - buffering
      - copying
      - OS traps / user-level access
  - «Στυλ» επικοινωνιών
    - Συγχρονισμένο ή τύπου rendezvous (το send() δεν ολοκληρώνεται αν ο παραλήπτης δεν έχει ξεκινήσει το receive())
    - Ασύγχρονο ή buffered
    - Blocking / non-blocking
    - Private / collective

Σύγχρονη επικοινωνία (τύπου rendezvous)

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



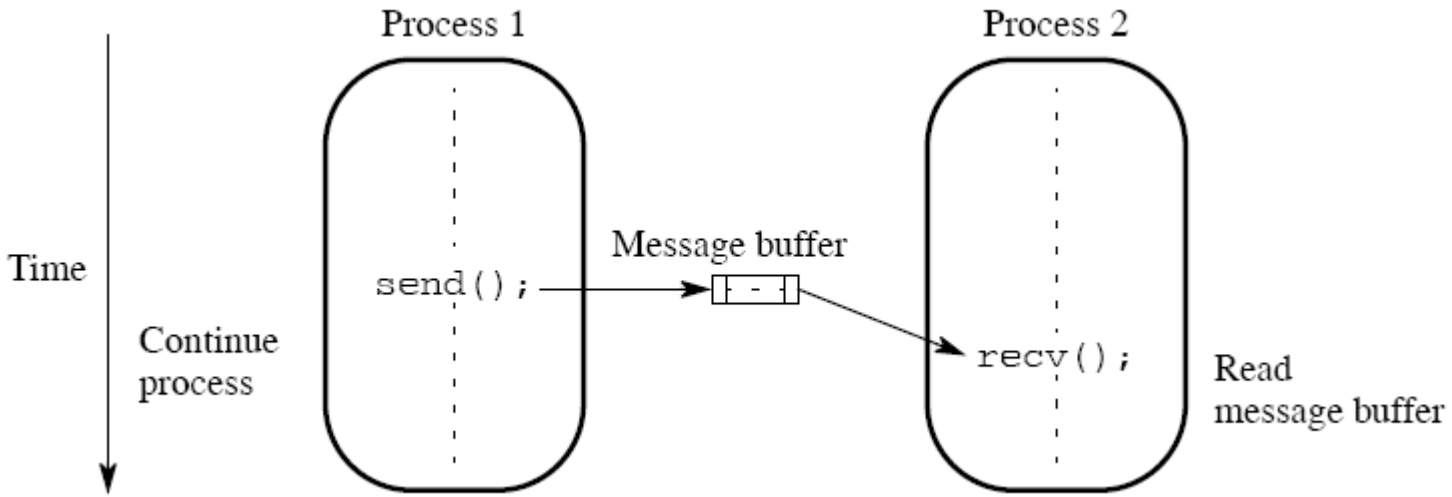
(a) When `send()` occurs before `recv()`



(b) When `recv()` occurs before `send()`

«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)



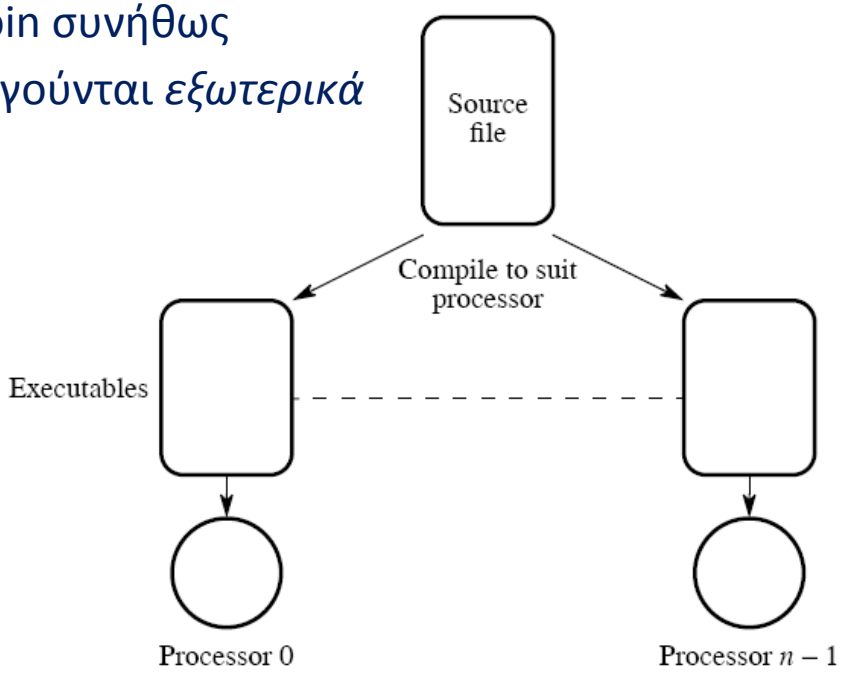
MPI

Προγραμματίζοντας με μεταβίβαση μηνυμάτων

- Δεν υπάρχουν κοινές μεταβλητές
  - διεργασίες και
  - μηνύματα
- Θεωρείται το πιο δύσκολο
- Μπορεί, όμως, να γίνει πιο αποδοτικό
- Κατάλληλο για cluster computing
  - Παράλληλος υπολογιστής του «φτωχού»
- Πολλές επιλογές… (sockets? pvm? …?)
  - … η εξής μία: 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

Παράδειγμα: υπολογισμός του π

```
#include <mpi.h>

main(int argc, char *argv[])
{
  float      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);

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

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

  MPI_Finalize();
}
```

Υπολογισμός του π

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

```
/* Συλλογή αποτελεσμάτων */
if (myid == 0)
{
  for (i = 1; i < nproc; i++)
  {
     MPI_Recv(&temp, 1, MPI_FLOAT, i, 0,
              MPI_COMM_WORLD, &status);
     result += temp;
  }
  printf("pi = %f\n", result);
}
```



```
/* Συλλογή αποτελεσμάτων */
if (myid == 0)
{
  for (i = 1; i < nproc; i++)
  {
     MPI_Recv(&temp, 1, MPI_FLOAT, MPI_ANY_SOURCE,
              0, MPI_COMM_WORLD, &status);
     result += temp;
  }
  printf("pi = %f\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!)

Τελικός κώδικας: πίνακας επί διάνυσμα (I)

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

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

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

    initialize_elements(A, v);       /* Κάποια αρχικοποίηση */
    for (i = 1; i < numprocs(); i++)
    {
      MPI_Send(A[i*WORK], WORK*N, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
      MPI_Send(v, N, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
    }
    for (i = 0; i < WORK; i++)
    {
      for (sum = 0.0, j = 0; j < N; j++)
        sum += A[i][j]*v[j];
      res[i] = sum;
    }
    /* Συλλογή των στοιχείων του αποτελέσματος */
    for (i = WORK; i < WORK*nprocs; i++)
    {
      MPI_Recv(&sum, 1, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      res[ status.MPI_TAG ] = sum;        /* Εξαγωγή της ετικέτας */
    }
    show_result(res);                /* Παρουσίαση αποτελέσματος */
  }
  else  /* του if (myid == 0) */       /* Υπόλοιπες διεργασίες */
  {
    float B[WORK][N];

    MPI_Recv(B, WORK*N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(v, N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
    for (i = 0; i < WORK; i++)
    {
      for (sum = 0.0, j = 0; j < N; j++)
        sum += B[i][j]*v[j];
      /* Το sum έχει το (myid*WORK+i)-οστό στοιχείο του αποτελ. */
      MPI_Send(&sum, 1, MPI_FLOAT, 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:

```
{
  float B[WORK][N], mypart[N];

  MPI_Recv(B, WORK*N, MPI_FLOAT, 0, 0,
           MPI_COMM_WORLD, &status);
  MPI_Recv(v, N, MPI_FLOAT, 0, 0,
           MPI_COMM_WORLD, status);
  for (i = 0; i < WORK; i++)
  {
    for (j = 0; j < N; j++)
      sum += B[i][j]*v[j];
    mypart[i] = sum;
  }
  /* Το mypart περιέχει WORK στοιχεία,
     αρχίζοντας από το (myid*WORK)-οστό */
  MPI_Send(mypart, WORK, MPI_FLOAT, 0,
           myid*WORK, MPI_COMM_WORLD);
}
```

Διεργασία 0:

```
if (myid == 0)
{
  ...
  /* Συλλογή στοιχείων του αποτελέσματος */
  for (i = 1; i < nprocs; i++)
  {
    /* Παραλαβή και εξαγωγή WORK στοιχείων */
    MPI_Recv(v, WORK, MPI_FLOAT, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    for (j = 0; j < WORK; j++)
      res[ j + status.MPI_TAG ] = v[j];
  }
  show_result(res);
}
```