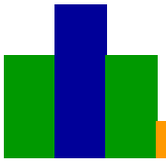


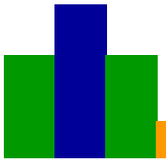
Lecţia 6:

Calcul paralel: Comunicare cu mesaje; Limbajul MPI

v1.0

Gheorghe Stefanescu — Universitatea Bucureşti

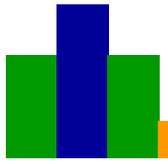
Metode de Dezvoltare Software, Sem.2
Februarie 2007— Iunie 2007



Calcul paralel: Comunicare cu mesaje

Cuprins:

- *Generalităţi*
- Tehnicii de paralelizare
- Sisteme distribuite
- Limbajul MPI
- Concluzii, diverse, etc.

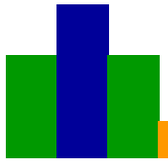


Opţiuni de programare paralelă

Programarea unui cluster/multi-computer ce *comunică cu mesaje (message-passing)* se poate face astfel:

- Proiectând un limbaj de programare paralelă *special*
  (e.g., OCCAM pentru transputere)
- *Extinzând* sintaxa unui limbaj secvenţial de nivel înalt cu proceduri de message-passing
  (e.g., CC+, FORTRAN M)
- Utilizând un limbaj secvenţial de nivel înalt şi dezvoltând o *bibliotecă* externă cu proceduri de message-passing
  (e.g., MPI, PVM)

Altă opţiune ar fi să scriem programe secvenţiale şi să lasăm paralelizarea în seama *compilatorului* pentru a produce un program paralel.



..Opţiuni de programare paralelă

Noi urmăm a 3-a cale. Atunci trebuie spus:

- *ce procese* se execută
- *când se trimit mesaje* între procese concurente
- *ce se trimite* în mesaje

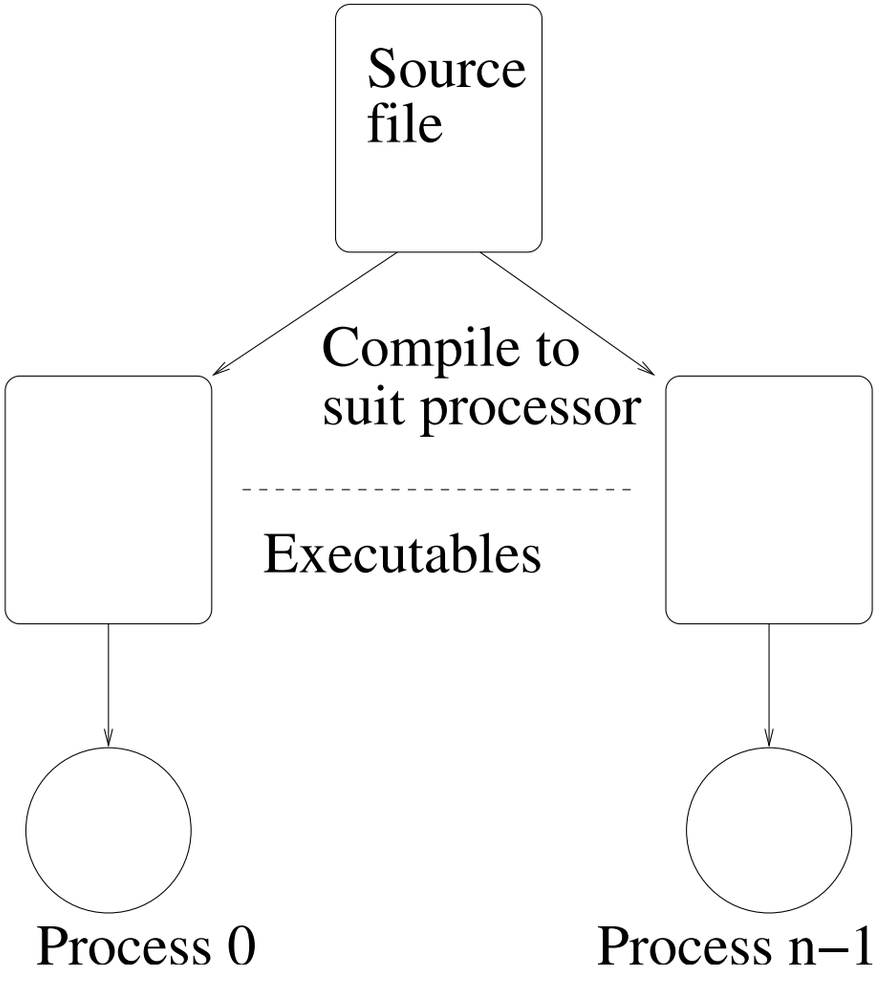
In mare, trebuie dezvoltate două metode pentru aceste sisteme:

- o metodă de *creare de procese separate* pentru execuţia pe diferite calculatoare
- o metodă de *trimitere şi recepţionare de mesaje*

SPMD (Single Program/Multiple Data)

In acest caz, *diverse procese* sunt specificate într-un *singur program*. In program, există structuri de control care permit mularea codului pe procese [i.e., selecţia unei părţi din program în funcţie de identitatea procesului].

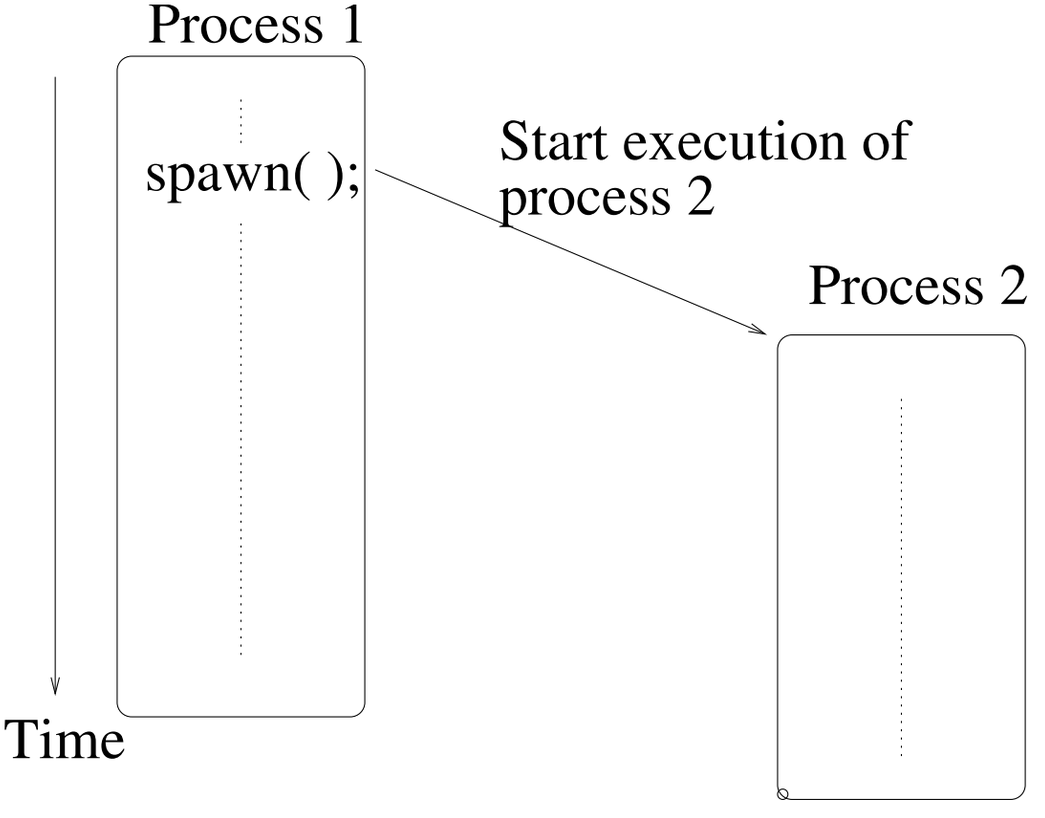
Caracteristici de bază:
—de obicei, procesele se crează *static*
—modelul de bază este MPI



MPMD (Multiple Program/Multiple Data)

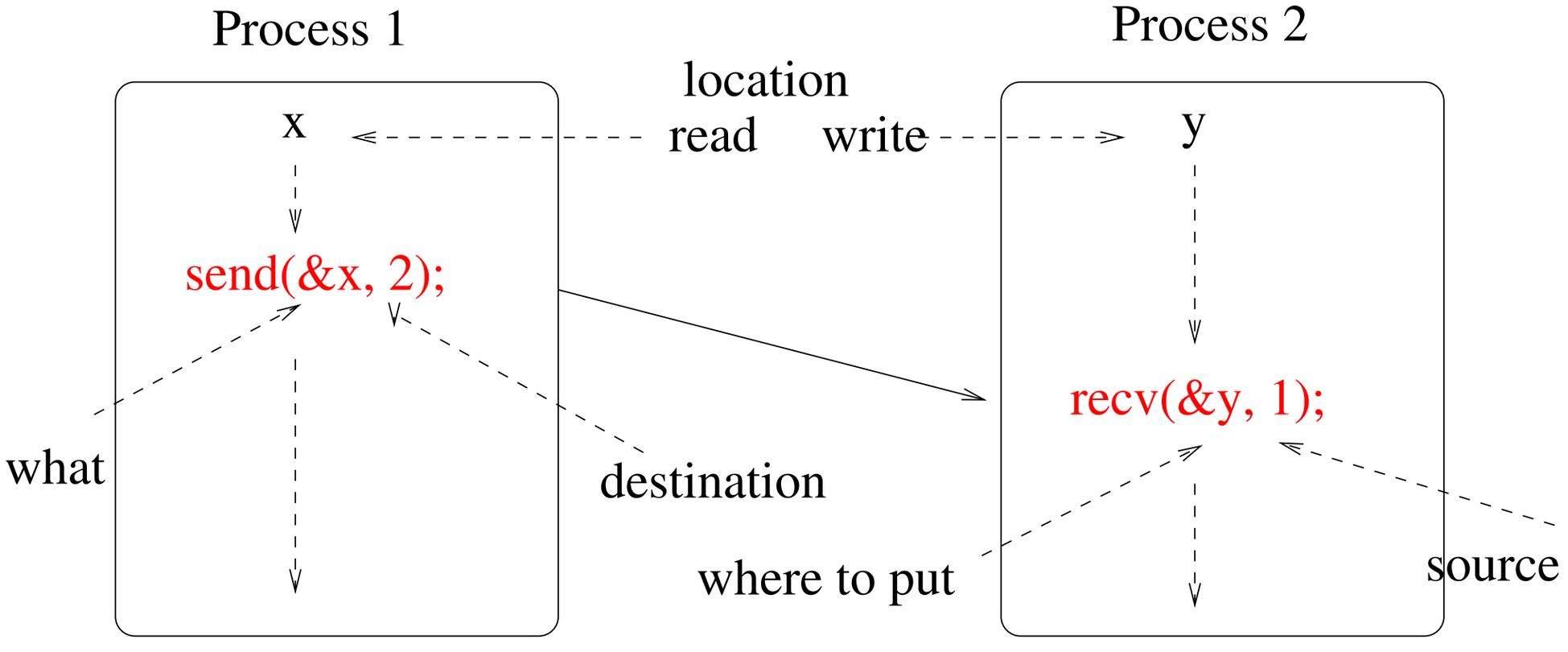
In acest caz, sunt scrise *programe separate* pentru fiecare procesor. De regulă, se aplică o metodă *master-slave*: un singur proces execută programul "master", restul proceselor fiind lansate de procesul master.

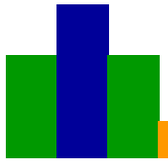
Caracteristici de bază:
—de obicei, procesele se crează *dinamic*
—modelul de bază este PVM



Rutinele send-receive de bază

Trimiterea de mesaje între procese cu rutinele `send()` şi `recv()`





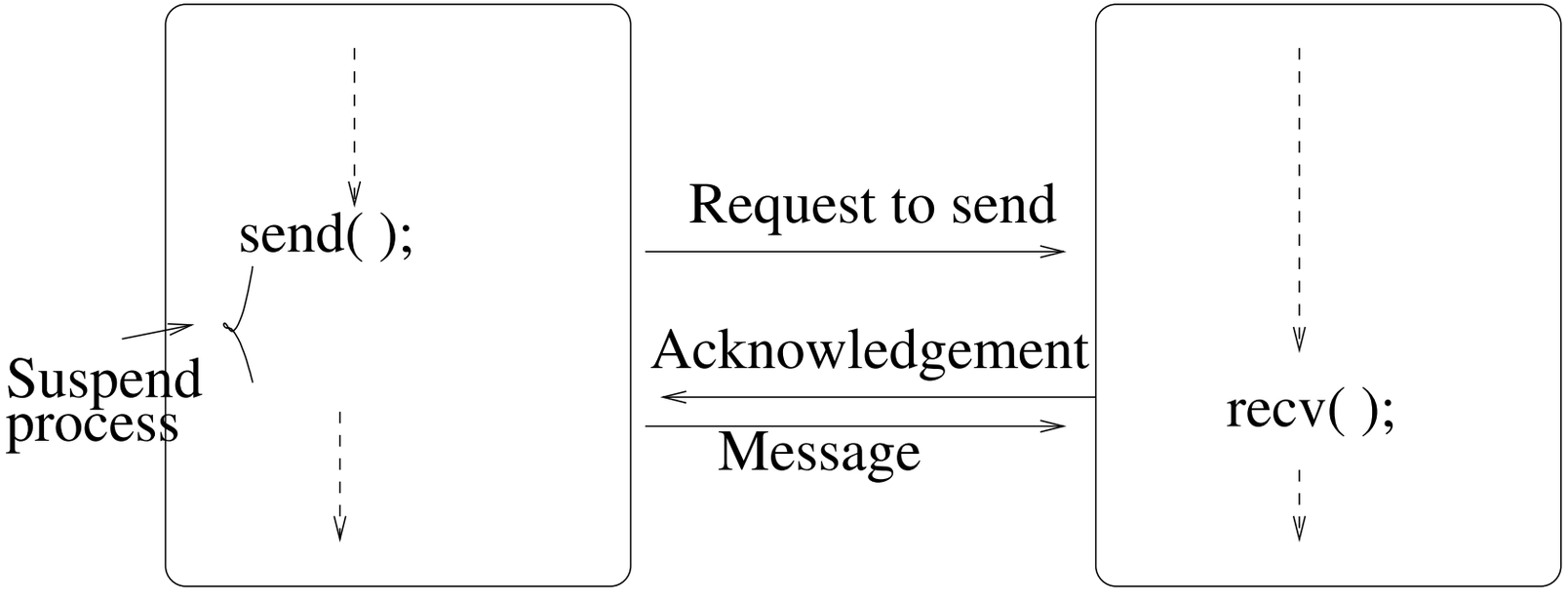
Message-passing sincron

- Rutinele de *message-passing sincron* se termină cu *return* când s-a terminat transferul mesajul.
- Nu este nevoie de buffer pentru memorare.
  - Rutinele de *send sincrone* vor aştepta până ce mesajul a fost complet acceptat de procesul care îl recepţionează.
  - Rutinele de *receive sincrone* aşteaptă până ce mesajul care trebuie primit soseşte integral.
- Rutinele sincrone execută două acţiuni de bază:
  - *transferă datele* şi
  - *sincronizează* procesele

..Message-passing sincron

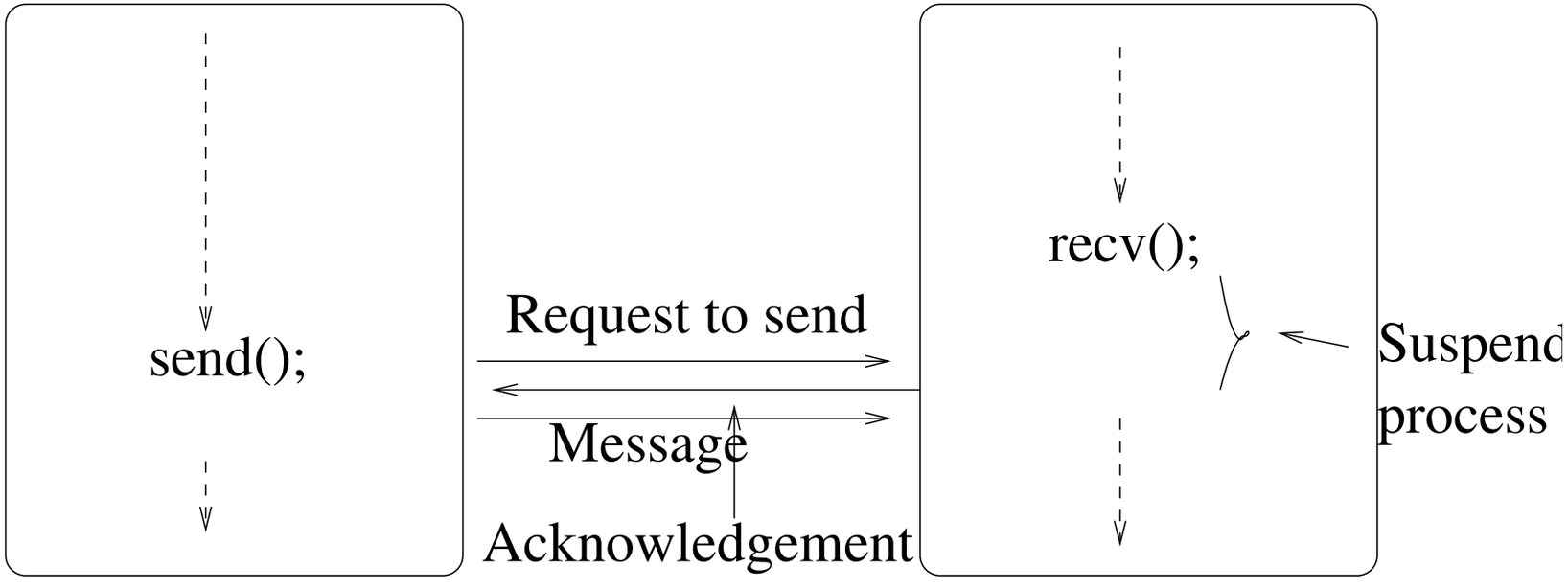
Protocolul este ilustrat în figurile următoare:

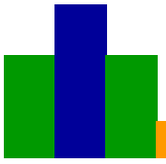
- Cazul 1: `Process 1` ajunge la `send()` înainte ca `Process 2` să ajungă la `recv()`:



..Message-passing sincron

- Cazul 2: `Process 1` ajunge la `send()` după ce `Process 2` ajunge la `recv()`:





Message-passing blocant şi ne-blocant

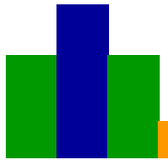
Blocant:

- acest termen este folosit pentru rutine din care *nu se returnează* înainte ca transferul să fie *complet*;
- mai precis, rutinele *blochează* procesul şi nu se poate continua codul;
- în genere, termenii *sincron* şi *blocant* sunt sinonimi;

Ne-blocant

- acest termen este folosit pentru rutine din care se returnează, *indiferent* dacă mesajul a fost recepţionat ori nu;

Notă: Aceşti termeni generali au un sens uşor modificat în MPI, vezi mai jos.



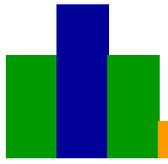
Definiţie MPI: blocant/ne-blocant

Blocant

- se face return după ce *acţiunile locale de trimitere s-au terminat*, deşi mesajul poate să nu fie încă transferat complet
- exemplu: pentru `send()` se face return după ce datele au fost puse în buffer pentru a fi trimise

Ne-blocant

- se face return imediat
- se presupune ca *memeoria cu datele* care se vor trimite *nu se modifică* prin instrucţiuni ulterioare înainte de completarea transferului (este datoria programatorului să asigure acest lucru)



Etichete pentru mesaje

O *etichetă de mesaj (tag)* este o informaţie auxiliară care diferenţiază mesajele.

Exemplu:

```
⋮                          ⋮
send(&x,2,5);              recv(&x,1,5);
⋮                          ⋮
(process 1)                (process 2)
```

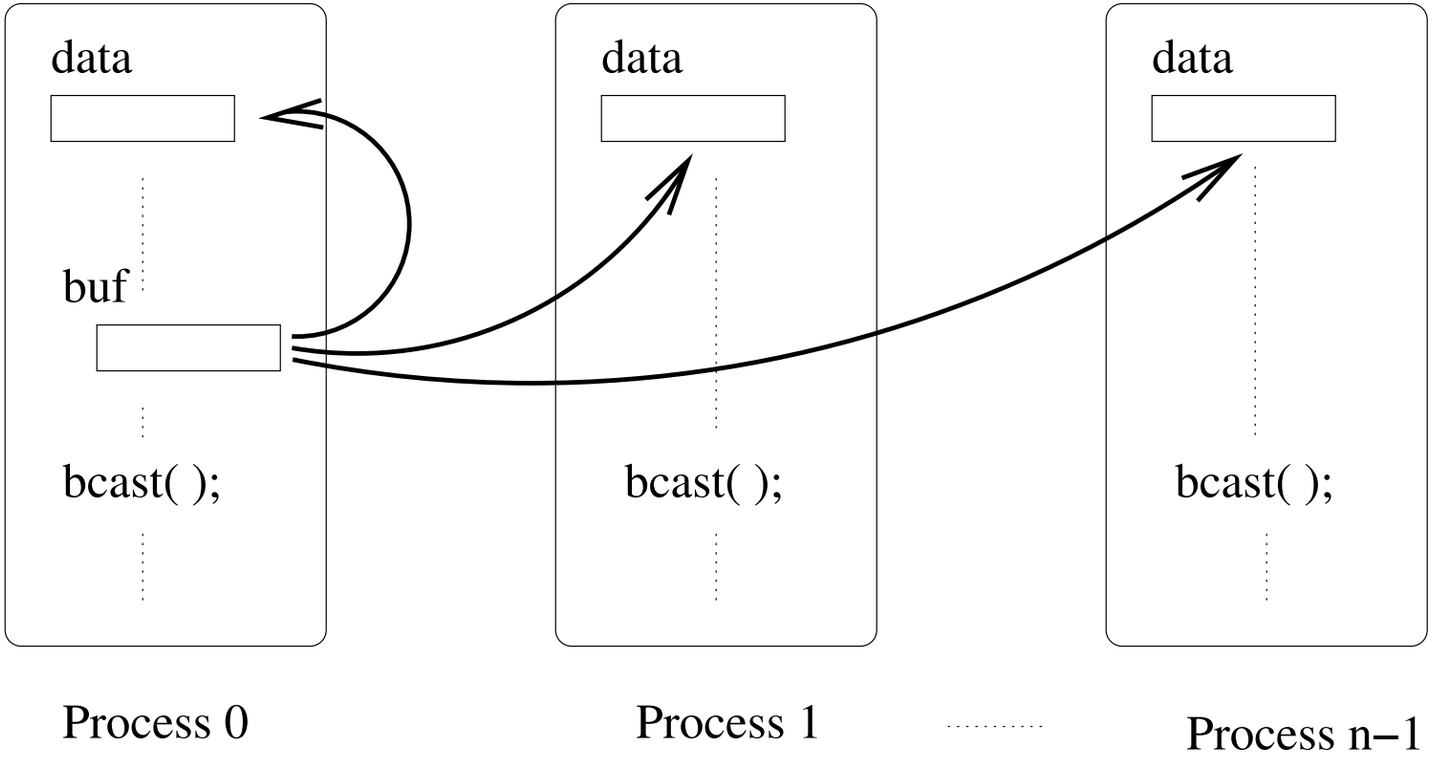
Eticheta 5 este utilizată pentru a cupla intrucţiunea send din `process 1` cu intrucţiunea receive din `process 2`.

Notă: Dacă nu trebuie o astfel de cuplare specifică, atunci se poate folosi o eticletă generică (*wild card*), astfel ca instrucţiunea `recv()` să se cuplează cu *orice* `send()`

Broadcast

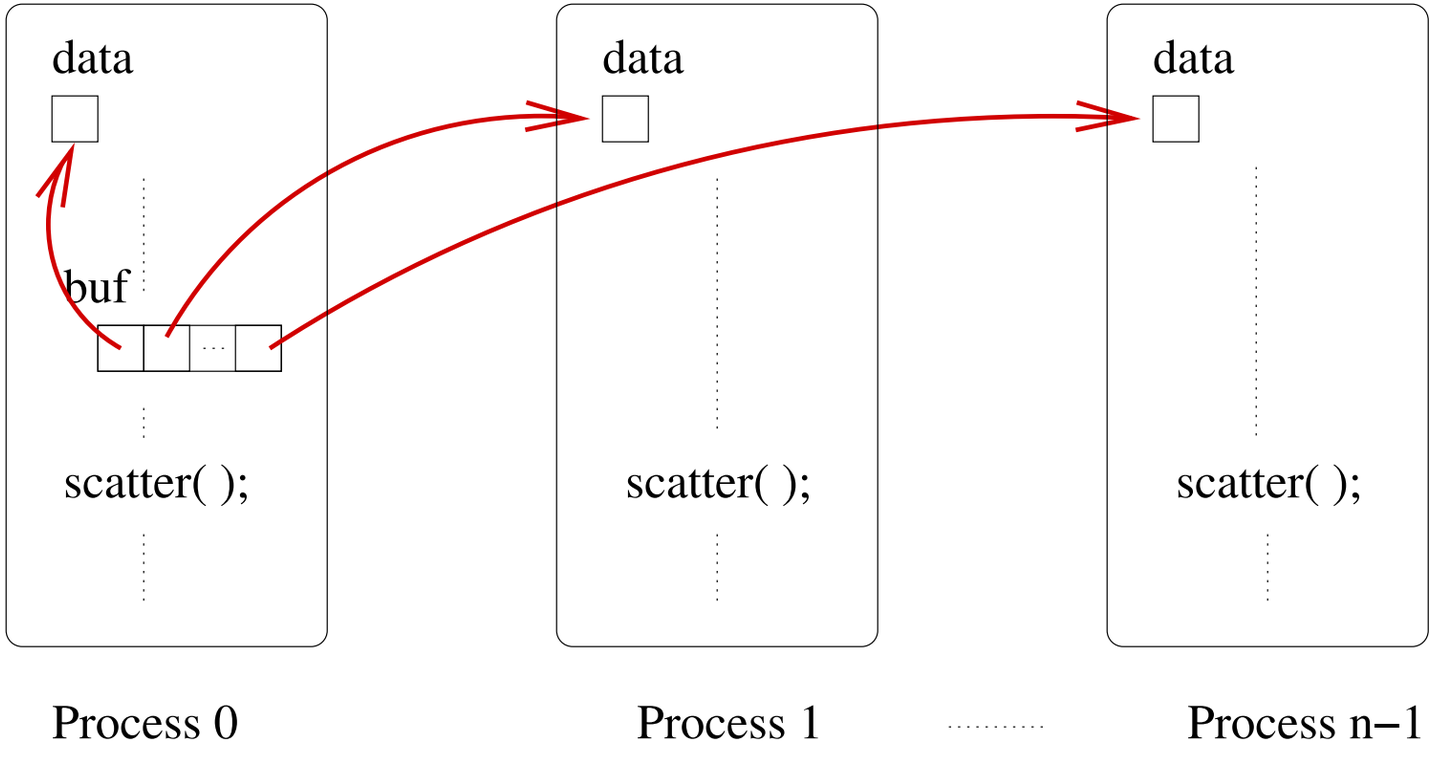
Broadcast-ul se foloseşte pentru a trimite un acelaşi mesaj *la toate* procesele care sunt flosite la problema în cauză.

Multicast-ul este similar, dar se trimite un acelaşi mesaj la toate procesele *dintr-un grup* specificat de procese.



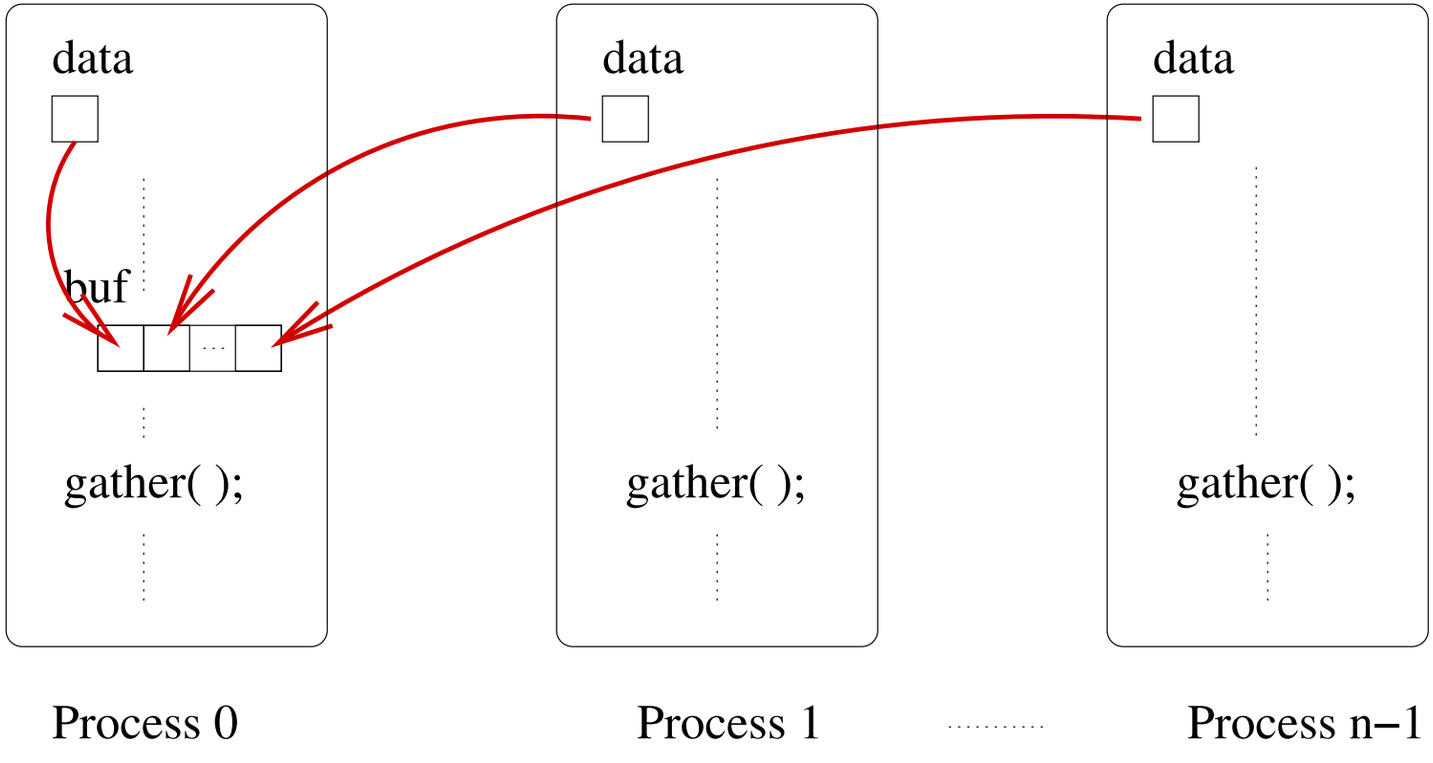
Scatter

Scatter-ul este folosit pentru a trimite *fiecare element dintr-un vector* de date de la procesul transmiţător la procesele care le recepţionează (anume, data din locaţia *i* merge la procesul *i*).



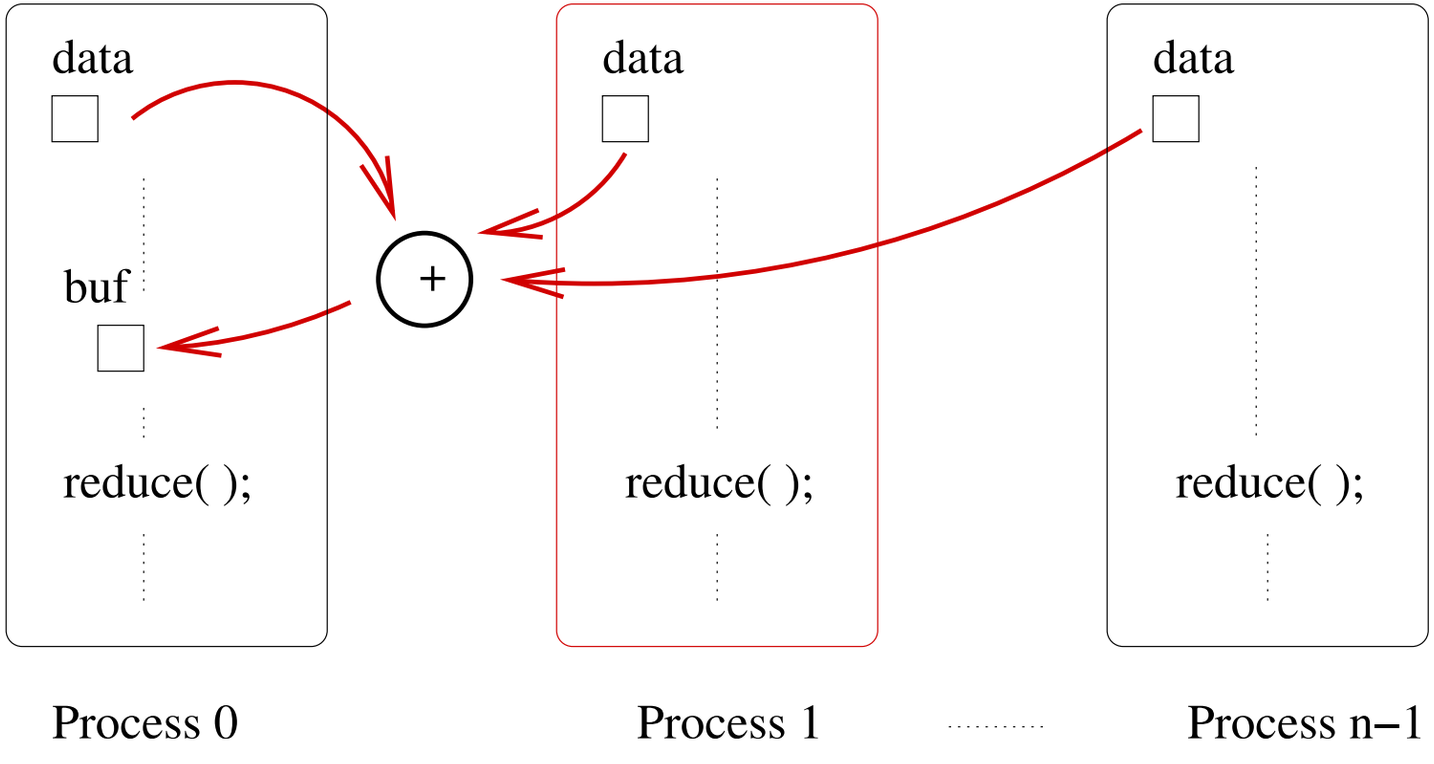
Gather

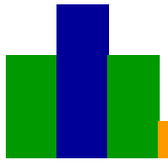
Gather-ul este operaţia opusă: procesul care recepţionează *colectează într-un vector* datele ce vin de la procese separate (data de la procesul *i* merge în locaţia *i*).



Reduce

Reduce: combină rutina `gather()` cu o operaţie aritmetică ori logică: procesul de recepţie *colectează* datele, *aplică* operaţia, şi *salvează* rezultatul în memoria sa.

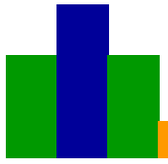




Software tools

PVM (Parallel Virtual Machine)

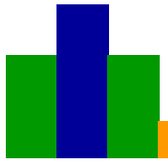
- prima abordare *larg acceptată* pentru clustere cu staţii de lucru ori pe multi-computere.
- se poate folosi pentru a rula programe atât pe platforme *omogene* cât şi *eterogene*
- are o colecţie de rutine ce pot fi folosite cu programe din *C* ori *FORTRAN*
- este *free*



..Software tools

MPI (Message Passing Interface)

- este un *standard* dezvoltat de un grup de specialişti din academie şi industrie interesaţi în a face message-passing-ul mai *protabil*
- există câteva *impelemntări free*
  [o recomand pe cea din Chicago, `mpich`]

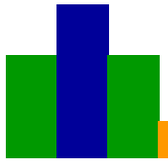


MPI

General: MPI este *standard* cu diverse implementări; se scrie *un singur program*, fiecare proces rulând propria sa copie;

Crearea şi execuţia proceselor: în genere, *nu se specifică*; la rulare, se specifică câte procese se vor folosi; în MPI, version 1, numai crearea statică de procese este permisă

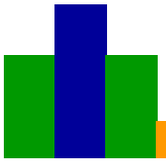
Comunicarea: se defineşte un *scope* pentru operaţia de comunicare; mulţimea proceselor care se folosesc se poate accesa cu variabila predefinită `MPI_COMM_WORLD`; fiecare proces are un rang unic, de la 0 la $n-1$ (unde n este numărul de procese); se pot defini alte grupuri de comunicare plecând de la acesta



..MPI

Model SPMD: Un program MPI are următoarea formă

```
main (int argc, char *argv[])
{
      MPI_Init(&argc,&argv);
      ⋮
/* find process rank */
      MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
      if (myrank == 0)
            /* master code */
      else
            /* slave code */
      ⋮
      MPI_Finalize();
}
```

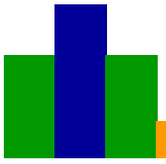


..MPI

Variabile globale şi locale: De regulă, orice declaraţie globală de variable va fi *duplicată* în fiecare proces; variabilele care nu se duplică trebuie să fie declarate în codul executat de proces

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0){
      int x,y;
⋮
} elseif (myrank == 1){
      int x,y;
⋮
}
```

(`x,y` din process 0 sunt diferite de `x,y` din process 1)



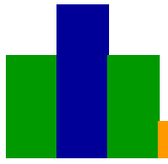
..MPI

Comunicare point-to-point: se folosesc etichete şi wild-card-uri (`MPI_ANY_TAG`, ori `MPI_ANY_SOURCE`)

Rutine blocante: returnează când sunt local complete, i.e., când locaţia folosită pentru mesaj poate fi folosită din nou *fără a afecta mesajul* trimis;

`MPI_Send(buf, count, datatype, dest, tag, comm)`
unde: `buf` - adresa buffer-ului send, `count` - numărul de elemente de trimis, `datatype` - tipul fiecărui element, `dest` - rangul procesului de destinaţie, `tag` - eticheta mesajului, `comm` - grupul comunicator

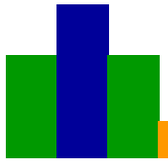
`MPI_Recv(buf, count, datatype, src, tag, comm, status)`
unde: `buf` - adresa buffer-ului receive, `count` - numărul maxim de elemente de primit, `datatype` - tipul fiecărui element, `src` - rangul procesului care a trimis mesajul, `tag` - eticheta mesajului, `comm` - grupul comunicator, `status` - starea după operaţie



..MPI

Exemplu (comunicare blocantă): Se trimite un întreg `x` de la process 0 la process 1

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0){
      int x;
      MPI_Send(&x,1,MPI_INT,1,73,MPI_COMM_WORLD);
} elseif (myrank == 1){
      int x;
      MPI_Recv(&x,1,MPI_INT,0,73,MPI_COMM_WORLD,status);
}
```

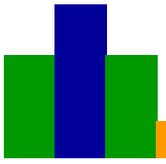


..MPI

Comunicare ne-blocanta: `MPI_Isend()` şi `MPI_Irecv()` - face return "*i*mediat", chiar dacă comunicarea nu este sigură; trebuie folosită în combinaţie cu `MPI_Wait()` şi `MPI_Test()` spre a asigura o comunicare sigură.

Exemplu (comunicare ne-blocantă):

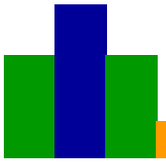
```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0){
        int x;
        MPI_Isend(&x,1,MPI_INT,1,73,MPI_COMM_WORLD,req1);
        compute();
        MPI_Wait(req1, status);
} elseif (myrank == 1){
        int x;
        MPI_Recv(&x,1,MPI_INT,0,73,MPI_COMM_WORLD,status);
}
```



..MPI

Moduri de comunicare pentru "send": Sunt 4 moduri:

1. *Modul send standard* - nu se presupune că rutina corespunzătoare receive este activată (spaţiul de buffer nu este definit în acest caz; dacă se alocă spaţiu de buffer, send-ul se poate completa înainte ca rutina receive corespunzătoare să fie apelată)
2. *Modul cu buffer* - rutina send poate fi apelată şi terminată înainte ca rutina receive corespunzătare să fie apelată (aici este necesar să se specifice spaţiul de buffer folosit)
3. *Modul sincron* - send şi receive trebuie să fie completate împreună (dar pot fi lansate oricând)
4. *Modul "ready"* - send poate fi lansat doar dacă s-a ajuns la rutina corespunzătoare receive (trebuie folosit cu grijă...)

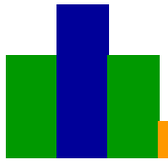


..MPI

Comunicare colectivă: Se aplică proceselor incluse într-un comunicator. Principalele operaţii sunt:

`MPI_Bcast()` - broadcast de la root la toate celelalte procese
`MPI_Gather()` - strânge valorile de la procesele dintr-un group
`MPI_Scather()` - distribuie părţi din buffer la diverse procese
`MPI_Alltoall()` - trimite date de la toate procesele la toate procesele
`MPI_Reduce()` - colectează şi combină valorile de la procese
`MPI_Reduce_scatter()` - combină valori şi le distribuie

Barrier: Se poate folosi pentru a sincroniza procesele oprindu-le până ce toate au ajuns la barieră

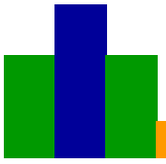


Exemplu de program MPI

Ilustrăm stilul de programare MPI cu un exemplu simplu: *adunăm numere* dintr-un fisier folosind mai multe procese.

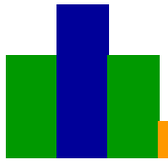
Folosim o metodă master-slave:

- Procesul master (process 0) detectează numărul de procese din comunicator, citeşte datele din fişier şi le trimite la toate procesele (prin broadcast).
- Fiecare proces (incluzând master-ul) identifică porţiunea sa de date şi le adună.
- Master-ul colectează sumele parţiale de la procese şi le adună (folosind instrucţiunea `reduce`) şi, în final, printează rezultatul final.



..Exemplu (program MPI)

```
01  #include "mpi.h"
02  #includes <stdio.h>
03  #include <math.h>
04  #define MAXSIZE 1000
05  void main(int argc, char *argv){
06        int myid, numprocs;
07        int data[MAXSIZE], i, x, low, high, myresult, result;
08        char fn[255];
09        char *fp;
10        MPI_Init(&argc,&argv);
11        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
12        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
13        if (myid == 0){
14              strcpy(fn,getenv("HOME"));
15              strcat(fn,"/MPI/rand_data.txt");
16              if((fp = fopen(fn,"r")) == NULL){
17                    printf("Can't open the input file %s\n\n",fn);
18                    exit(1);
19              }
```



..Exemplu (program MPI)

```
20              for (i=0; i<MAXSIZE; i++) fscanf(fp,"%d",&data[i]);
21          }
22          /* Broadcast data */
23          MPI_Bcast(data,MAXSIZE,MPI_INT,0,MPI_COMM_WORLD);
24          /* Add my portion of data */
25          x = n / nproc;
26          low = myid * x;
27          high = low + x;
28          for (i=low; i<high; i++)
29              myresult += data[i];
30          printf("I got %d from %d\n", myresult,myid);
31          /* Compute global sum */
32          MPI_Reduce(&myresult,&result,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
33          if (myid == 0) printf("The sum is %d.\n",result);
34          MPI_Finalize();
35      }
```

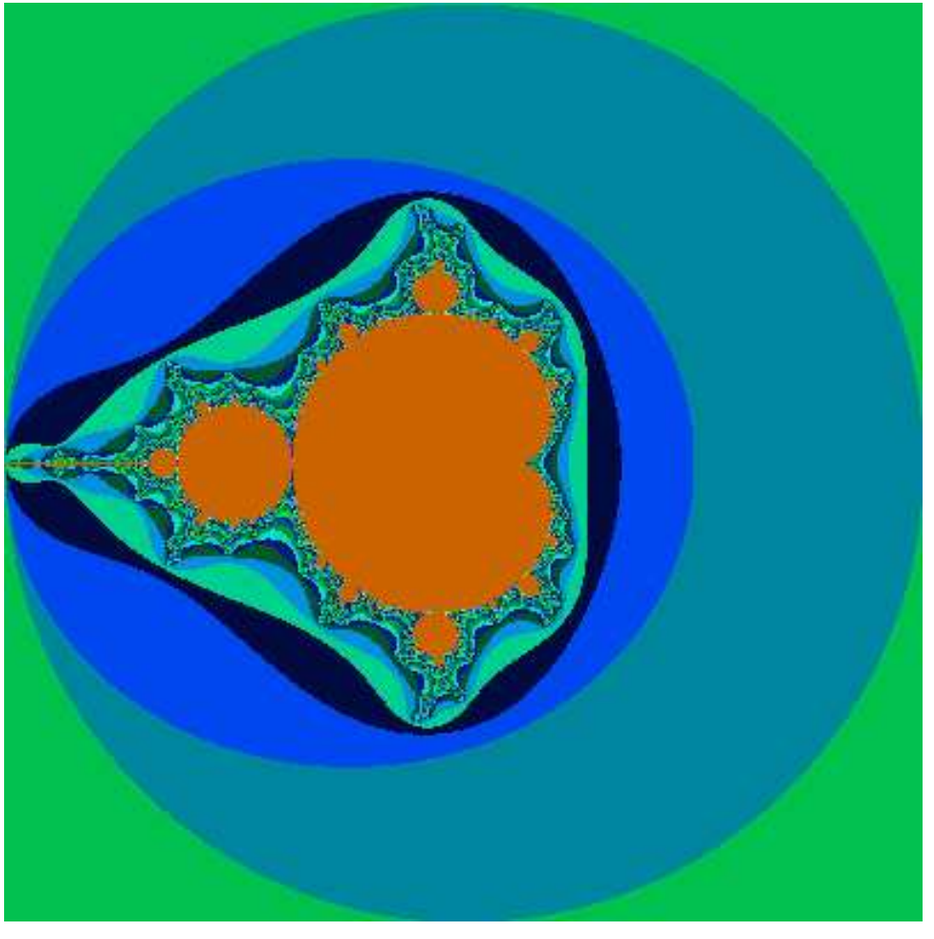
Exemplu: Multimi Mandelbrot

Mulţimi Mandelbrot: Se foloseşte transformarea

$$z \mapsto z^2 + c$$

pe numere complexe. Pentru un c dat se iterează începând cu 0 până ce

—modulul numărului este mai mare de 2, ori
—numărul de iterări a atins o limită maximă.



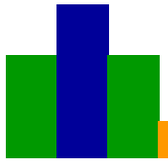
Notă: Este un exemplu de calcul intensiv pentru fiecare pixel. Culorile din desen reprezintă numărul de paşi necesari pentru a obţine un modul mai mare ca 2.

..Multimi Mandelbrot

Un program sevenţial

```
structure complex{
      float real;
      float imag;
}

int calcPixel(complex c){
      int count, max;
      complex z;
      float tmp, lengthSq;
      max = 256;
      z.real = 0; z.imag = 0;
      count = 0;
      do{
            tmp = z.real * z.real - r.imag * z.imag + c.real;
            z.imag = 2 * z.real * z.imag - c.imag;
            z.real = tmp;
            lengthSq = z.real * z.real + z.imag * z.imag;
            count++;
      } while (lengthSq < 4.0) && (count < max));
      return count;
}
```



..Multimi Mandelbrot

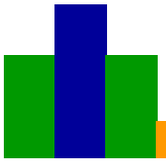
Versiune paralelă (cu asignare statică a job-urilor):

Master code

```
for (i=0,row=0; i<48; i++,row=row+10)
      send(row,Pi);
for (i=0; i<(480*640); i++){
      recv(c,color,Pany);
      display(c,color);
}
```

Slave code (folosim factori de scalare pentru a se protrivi pe display, i.e., `scaleReal = (realMax-realMin)/dispWidth` şi similar pentru `scaleImg`)

```
recv(row,Pmaster);
for (x=0; x<dispWidth, x++)
      for (y=row; y<(row+10), y++){
            c.real = minReal + ((float)x * scaleReal);
            c.imag = minImag + ((float)y * scaleImag);
            color = calcPixel(c);
            send(c,color,Pmaster);
      }
```



..Mulţimi Mandelbrot

Analiză grosieră:

- Secvenţial: $t_s \leq max \times n = O(n)$
- Paralel ($p+1$ procese):

  —Comm-1: trimite numărul de linii fiecărui scalv

  $t_{comm1} = p(t_{startup} + t_{data})$

  —Calcul: sclavii calculează în paralel

  $t_{comp} \leq \frac{max \times n}{p}$

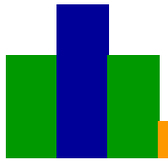
  —Comm-2: rezultatele se trimit la master [send individual]

  $t_{comm2} = \frac{n}{p}(t_{startup} + t_{data})$

  —Timp de execuţie total:

  $t_p \leq \frac{max \times n}{p} + (\frac{n}{p} + p)(t_{startup} + t_{data})$

Concluzie: Când *max este mare* primul factor este dominant şi speedup-ul (raportul timp secvenţial pe timp paralel) ajunge aproape de $p-1$.



..Multimi Mandelbrot

Observaţii pentru a îmbunătăţi eficienţa programului paralel:

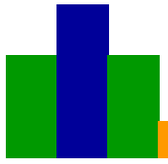
- Timpul de stabilire a comunicării (startup time) este în genere lung, deci este mai bine să aşteptăm până ce *toţi pixelii dintr-o linie* sunt calculaţi şi să trimitem o linie întreagă master-ului
- Timpii de calcul din diverse procese sclav pot fi diferite, deci este mai bine să se *distribuie job-urile dinamic*:
  —se alocă liniile sclavilor una câte una;
  —când un sclav este gata şi a returnat o linie calculată i se alocă o nouă linie pentru procesare.

..Mulţimi Mandelbrot

Versiune paralelă (cu job-uri alocate dinamic):

Cod pentru master

```
count = 0;
row = 0;
for (k=0; k < procNo; k++){
     send(row,P_k,dataTag);
     count++;
     row++;
}
do{
     recv(slave,r,color,P_any,resultTag);
     count--;
     if(row<dispHeight){
          send(row,P_slave,dataTag);
          count++;
          row++;
     } else
          send(row,P_slave,terminatorTag);
     rowRecv++;
     display(r,color);
} while(count>0);
```



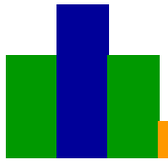
..Multimi Mandelbrot

Cod pentru sclavi

```
recv(y,Pmaster,sourceTag);
while(sourceTag == dataTag){
      c.imag = imagMin + ((float) y * scaleImag);
      for (x=0; x < dispWidth; x++){
            c.real = realMin + ((float) x * scaleReal);
            color[x] = calcPixel(c);
      }
      send(i,y,color,Pmaster,resultTag);
      recv(y,Pmaster,sourceTag);
}
```

(Variabila `count` *se foloseşte pentru a şti numărul de sclavi ocupaţi.)*

Notă: Pentru acest tip de algoritmi este mai utilă o evaluare empirică a timpului total de execuţie.



Divide-si-cucereste

Divide-si-cucereste este o strategie de partiţionare în care *subprce-sele au acelaşi format* ca şi problema mai mare.

Deci, procedura de partiţionare se poate aplica iterativ pentru a obţine probleme din ce în ce mai mici.

Adunarea unei liste de numere:

```
int add(list){
if (numbersOfElements(list) =< 2) return theirSum;
else{
      divide(list,list1,list2);
      return(add(list1) + add(list2));
     }
}
```

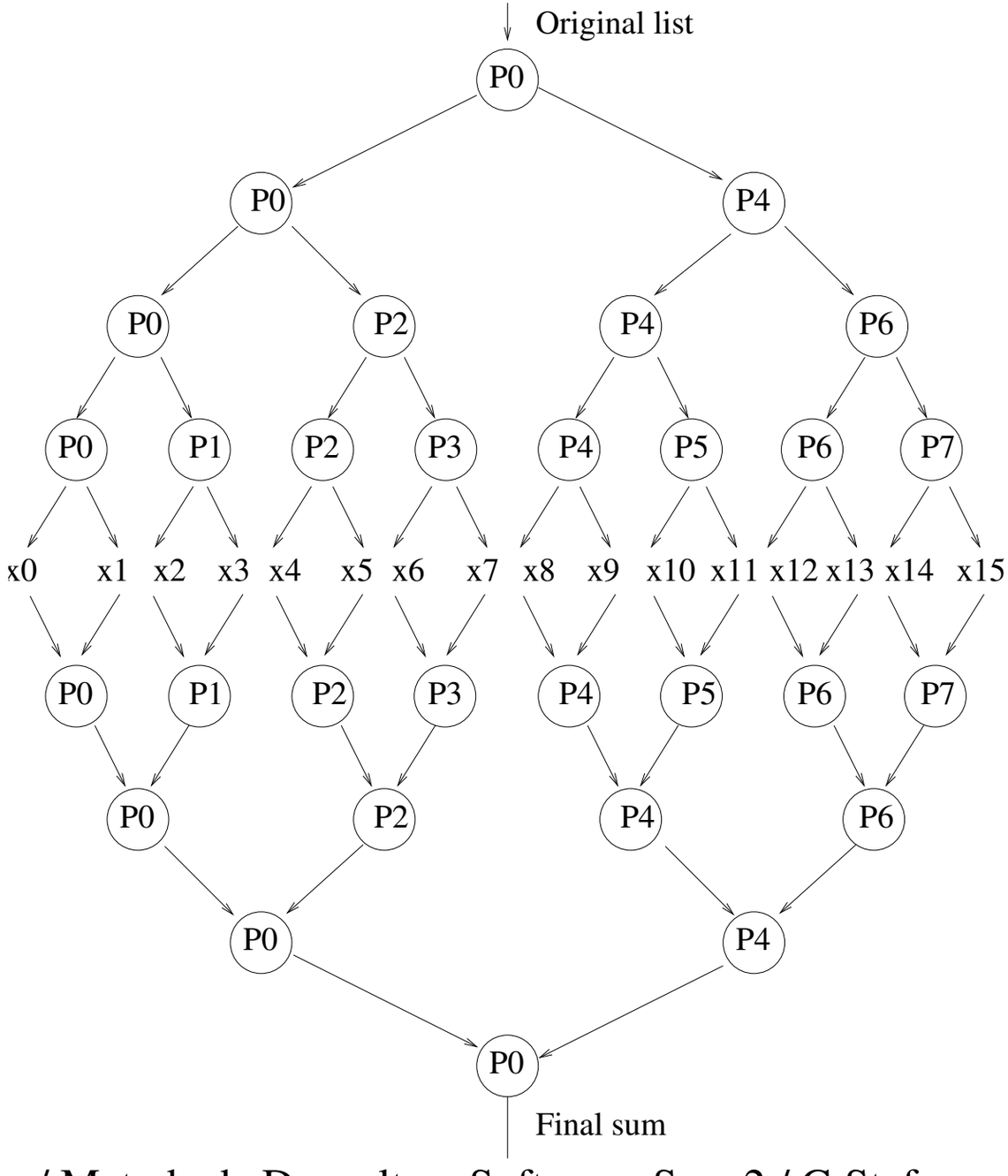
Notă: Este o procedură generală care se poate aplica fie secvenţial, fie paralel.

..Divide-si-cucereste

Implementare paralelă:

(a) In prima fază, lista de numere se divide recursiv până ce se obţin liste de 2 elemente.

(b) In faza următoare, rezultatele parţiale sunt colectate folosind o structură arborescentă opusă.

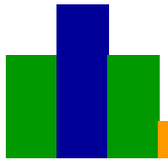


..Divide-si-cucereste

Cod paralel (Process P_i): Fie

$$k(i) = \begin{cases} \bullet\ r, \text{ dacă } i = 0 \text{ (ăi există } 2^r \text{ procese)} \\ \bullet\ \text{“cea mai mare putere a lui 2 care divide } i\text{”, altfel} \\ \quad \text{[i.e., } 2^{k(i)} \text{ divide } i, \text{ dar } 2^{k(i)+1} \text{ nu divide } i] \end{cases}$$

```
if (!(myRank == 0)) recv(list,P_{myRank-2^{k(i)}});
for(i=k-1; i>=0; i--){
     divide(list,list,list2);
     send(list2,P_{myRank+2^i});
}
partSum = sumOf(list);
for(i=0; i<k; i++){
     recv(partSum2,P_{myRank+2^i});
     partSum = partSum + partSum2;
}
if (!(myRank == 0)) send(partSum,P_{myRank-2^{k(i)}});
```



..Divide-si-cucereste

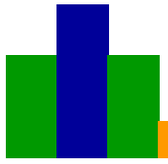
Exemple:

Process P0

```
divide(list,list,list2);
send(list2,P4);
divide(list,list,list2);
send(list2,P2);
divide(list,list,list2);
send(list2,P1);
partSum = sumOf(List);
recv(partSum2,P1);
partSum = partSum + partSum2;
recv(partSum2,P2);
partSum = partSum + partSum2;
recv(partSum2,P4);
partSum = partSum + partSum2;
```

Process P4 ($k(4) = 2$):

```
recv(list,P0);
divide(list,list,list2);
send(list2,P6);
divide(list,list,list2);
send(list2,P5);
partSum = sumOf(List);
recv(partSum2,P5);
partSum = partSum + partSum2;
recv(partSum2,P6);
partSum = partSum + partSum2;
send(partSum,P0);
```



..Divide-si-cucereste

Analiza: Să presupunem că $n = 2^k$ şi $p = 2^r$. Atunci:

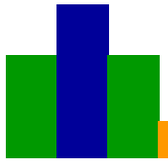
—diviziune: $t_{comm1} = t_{startup}\log_2 p + (n/2 + n/4 + \ldots + n/(2^r))t_{data}$
$= t_{startup}\log_2 p + [n(p-1)/p]t_{data}$

—colectare rezultate: $t_{comm2} = t_{startup}\log_2 p + t_{data}\log_2 p$

—calcul: $t_{comp} = n/p - 1 + \log_2 p$

—total:

$$t_p = 2t_{startup}\log_2 p + [\log_2 p + n(p-1)/p]t_{data} + n/p - 1 + \log_2 p$$



..Divide-si-cucereste

Concluzie:

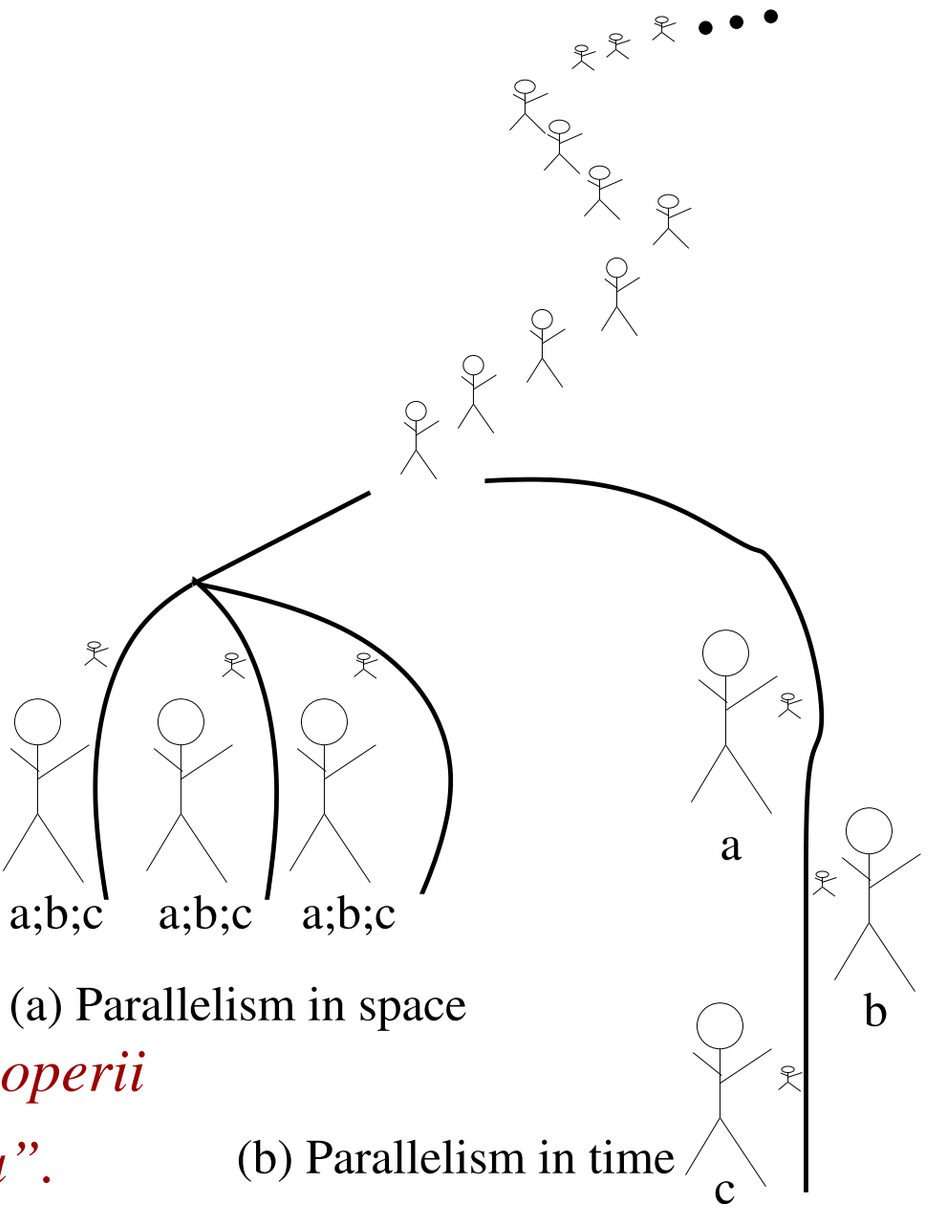
- Partea de calcul *descreşte*.
- Comunicarea este încă largă: *lineară* în mărimea datelor şi *log-arithmică* în numărul de procese.
- Metoda devine viabilă când partea de calcul consumă mult timp.

Notă: Să notăm ca sarcina nu este uniform distribuită pe procese.

Paralelism in spatiu si in time

Se pot identifica două tipuri de paralelism:
—*în spatiu* si
—*în timp*.

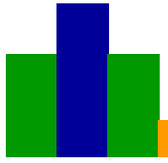
In primul caz (a), secvenţa completă de acţiuni `a;b;c;` este procesată de un singur proces, pe când în cazul (b) avem procese specializate pentru fiecare acţiune `a`, `b`, şi `c`.



(a) Parallelism in space

(b) Parallelism in time

Notă: In fapt, (b) este o ilustrare a descoperii lui Ford că "specializarea creşte eficienţa".



Aplicatii pipeline

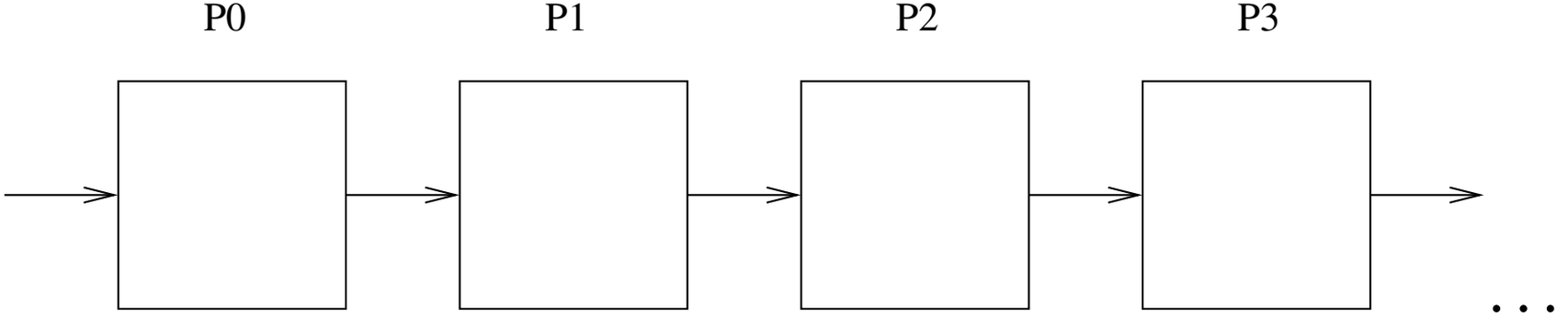
Aplicaţii:

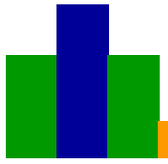
- Abordarea "*pipeline*" a fost multă vreme sursa principală de a ceşte *viteza procesoarelor*. (Curent, accentul se pune pe "instruction-level parallelism" şi "hyper-threading")
- Este o tehnică de bază în *arhitecturile paralele de tip data-flow*, propuse ca alternativă la arhitectura clasică Von Neumann.

Tehnica pipeline

In *tehnica pipeline*,

- Problema se divide într-o *serie de sarcini* care trebuie completate *una după alta*. [In fapt, asta este esenţa programării secvenţiale.]
- Apoi, *fiecare sarcină* este executată de un *proces separat* (ori procesor).





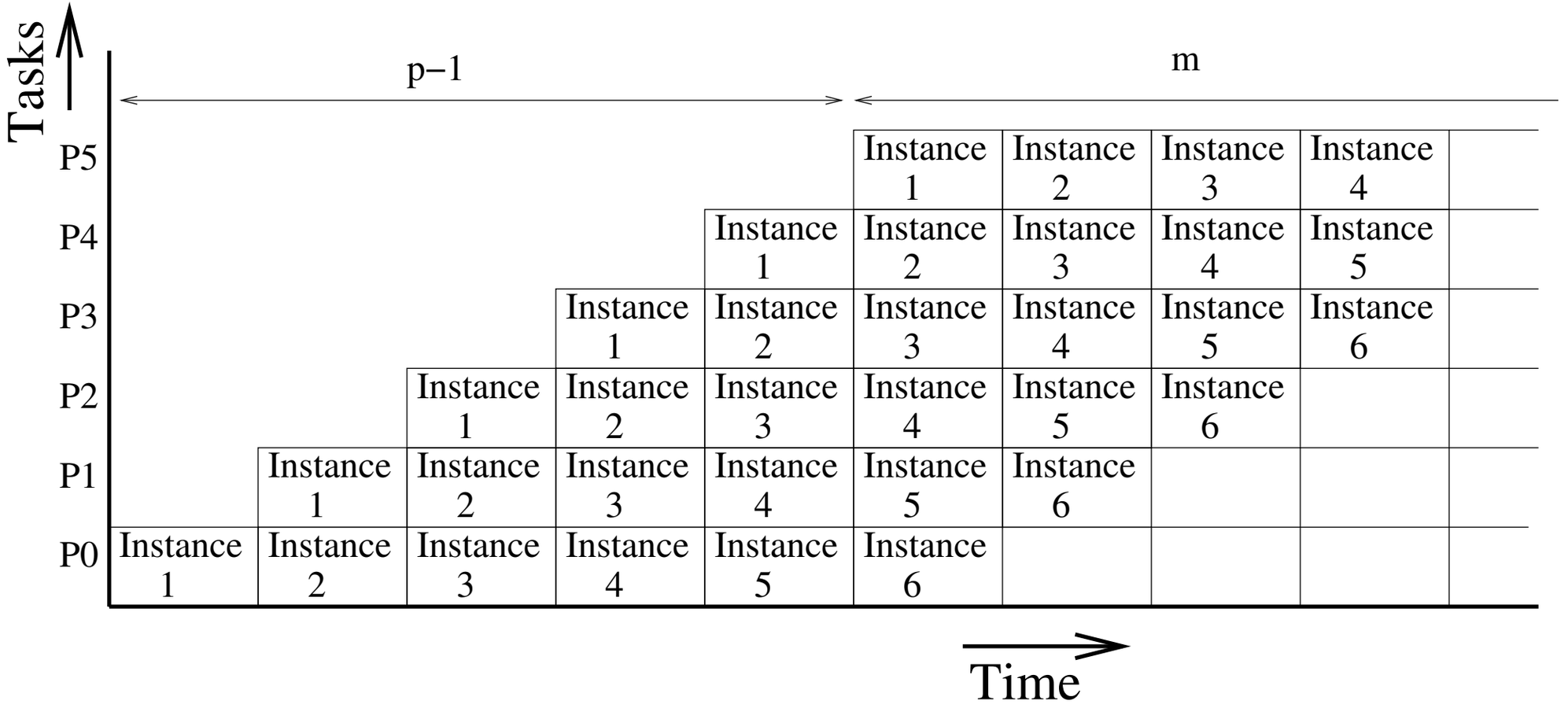
..Tehnica pipeline

Bună pentru: Dată împărţirea sarcinii în sarcini de executat secvenţial, tehnica pipeline *creşte viteza* în următoarele situaţii:

- **Tip 1**: Dacă *mai multe instanţe* ale unei singure probleme trebuie executate
- **Tip 2**: Dacă trebuie procesată *o serie de date*, fiecare necesitând mai multe operaţii
- **Tip 3**: Dacă informaţia privind lansarea următorului proces poate fi *pasată altor procese* înainte ca procesul curent să termine execuţia tuturor operaţiilor sale.

Diagrame spatiu-timp

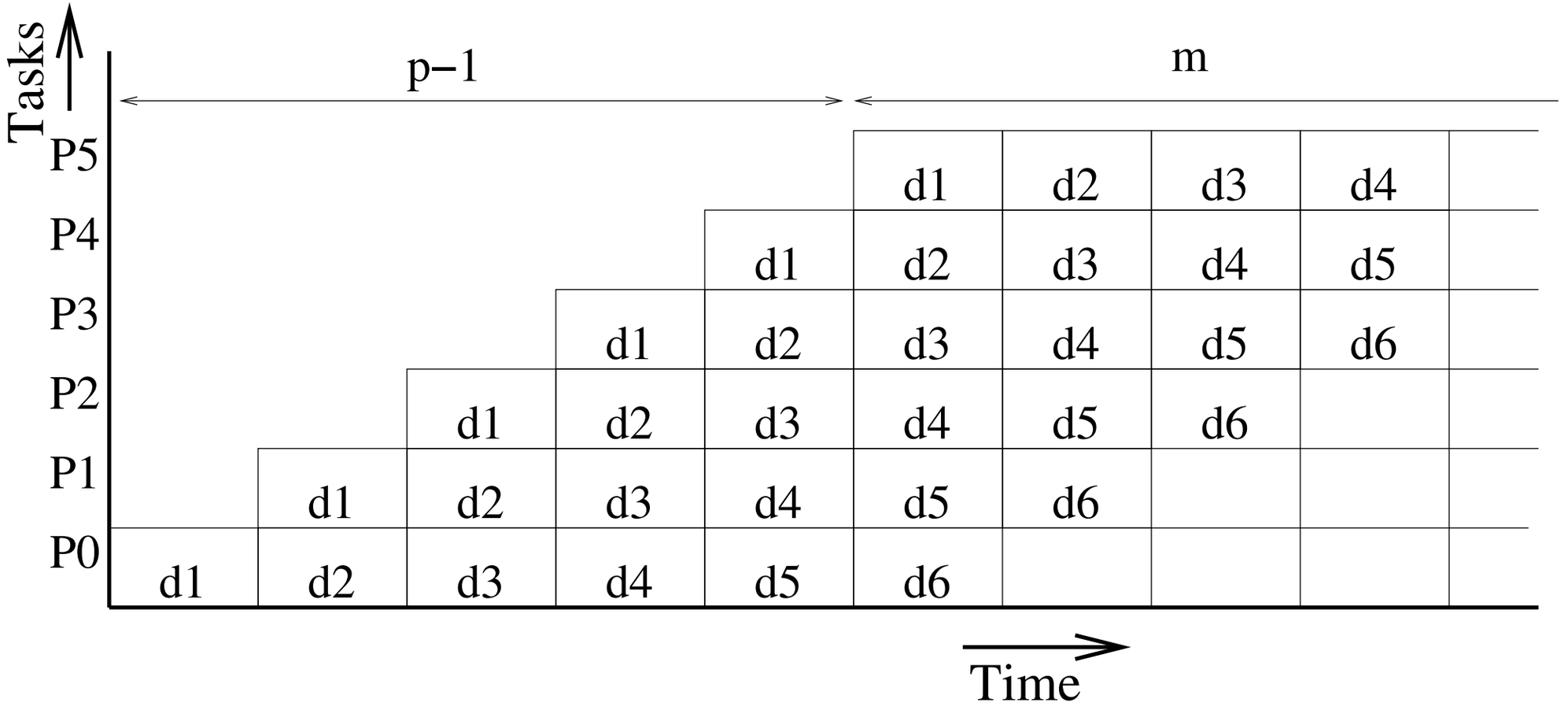
Tip 1: Instanţe multiple ale aceleaşi problme.



..Diagrame spatiu-timp

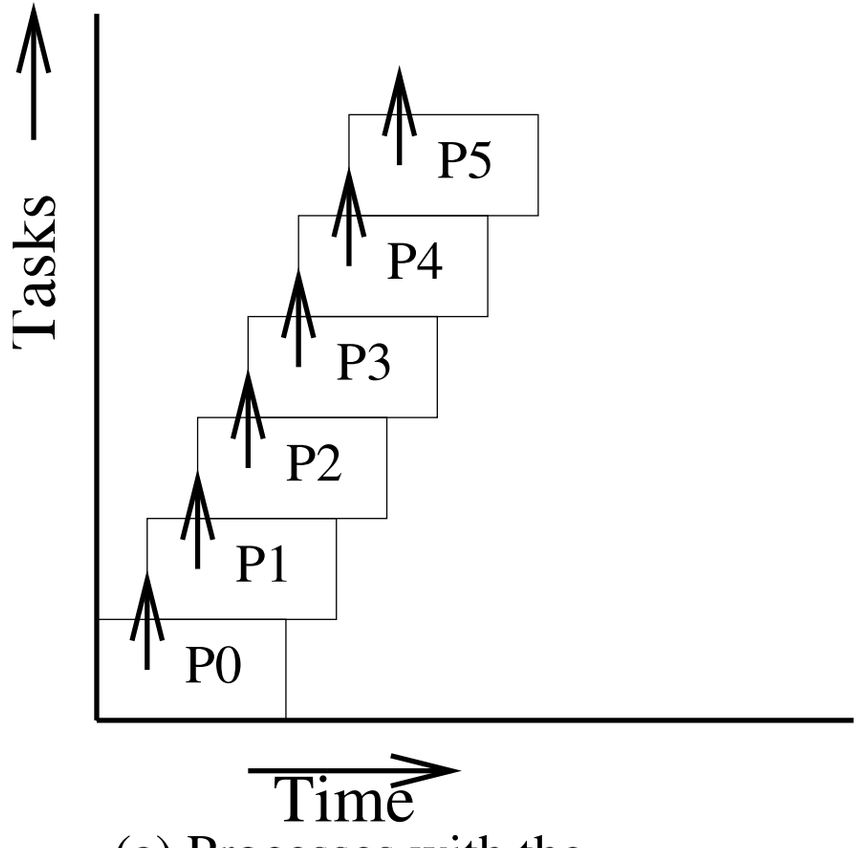
Tip 2: Strucură pipeline pentru date repetate

d6 d5 d4 d3 d2 d1 d0 → P0 → P1 → P2 → P3 → P4 → P5

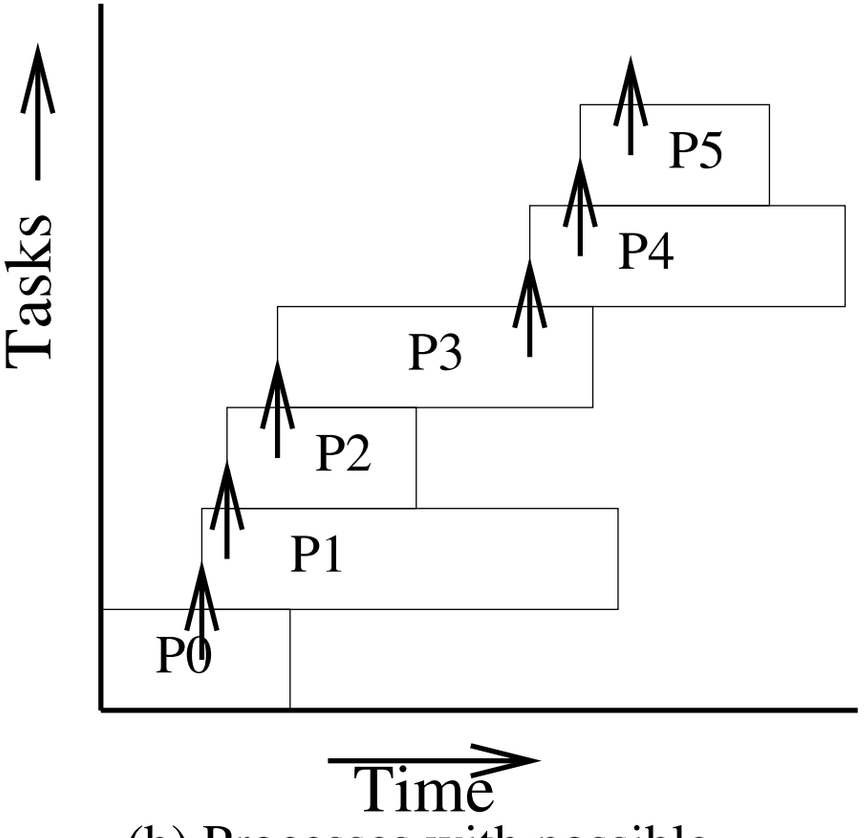


..Diagrame spatiu-timp

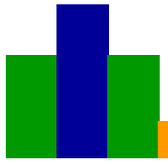
Tip 3: Procesare pipeline unde informaţia necesară pentru lansarea unui proces se trimite *înainte* ca procesul curent să-şi termine execuţia.



(a) Processes with the same execution time



(b) Processes with possible different execution time



Exemplu: Numere prime

Generează numere prime folsind ciurul lui Eratostene

Să presupunem că vrem sa aflăm *numerele prime* de la 2 la 20.
Introducem toate numerele

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20$$

primul număr 2 este prim; îl marcăm împreună cu toţi multiplii săi

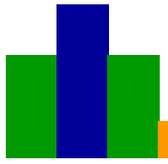
$$2, 3, \not4, 5, \not6, 7, \not8, 9, \not{10}, 11, \not{12}, 13, \not{14}, 15, \not{16}, 17, \not{18}, 19, \not{20}$$

primul număr nemarcat 3 este prim; îl marcăm împreună cu toţi multiplii săi

$$2, 3, \not4, 5, \not6, 7, \not8, \not9, \not{10}, 11, \not{12}, 13, \not{14}, \not{15}, \not{16}, 17, \not{18}, 19, \not{20}$$

... şi aşa mai departe.

Ca să aflăm toate numerele prime până la n repetăm procedura până la numerele prime apropiate de $\sqrt{n}$.

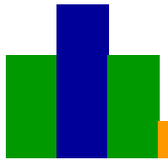


..Numere prime

Cod secvenţial:

```
for (i=2; i<n; i++)
    prime[i] = 1;
for (i=2; i<=sqrt(n); i++){
    if (prime[i] == 1){
        for (j=i+i; i<n; j=j+i)
            prime[j] = 0;
    }
}
```

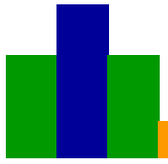
programul foloseşte un vector `prime` *astfel că în final* `prime[i]` *este 1 dacă* `i` *este prim, altfel este 0.*



..Numere prime

Cod paralel:

- O procedură de partiţionare poate fi destul de ineficientă.
- Soluţia pipeline dată aici este următoarea:
  - fiecare process *reţine primul număr primit*, să zicem p, ca număr prim şi
  - *trece mai departe* acele numere care *nu sunt multipli de* p.
- Procedura este mai eficientă căci evită marcarea multiplă a numerelor care au mai mulţi factori primi.



..Numere prime

Pseudo-codul pentru nucleul procesului P_i este

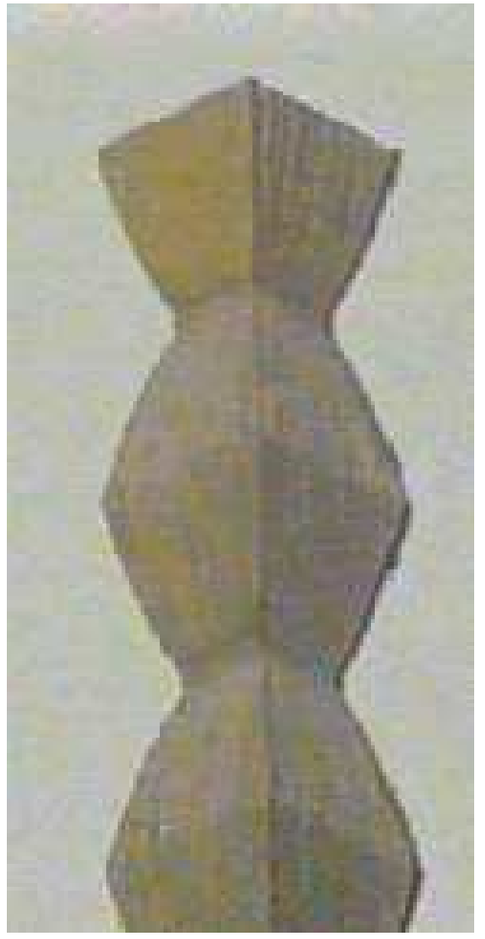
```
recv(&x,P_{i-1});
for (i=0; i<n; i++){
    recv(&number,P_{i-1});
    if (number == terminator) break;
    if ((number % x) != 0) send (&number P_{i+1});
}
```

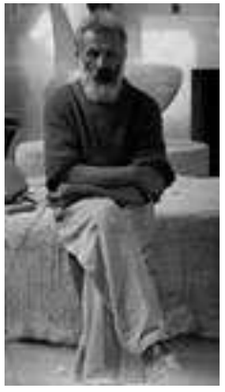
Notă: Avem nevoie de un mesaj special "terminator", căci numărul de paşi nu este ştiut apriori. Operatorul "%" este costisitor şi trebuie evitat.

Calcul sincron

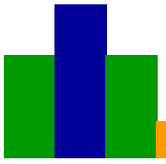
Calcul sincron:

- toate procesele se *sincronizează repetat* la anumite puncte
- este o clasă foarte improtantă de probleme
  (e.g., 70% dint-un set Caltech de aplicaţii la programarea paralelă)





Brancusi: Coloana infinitului



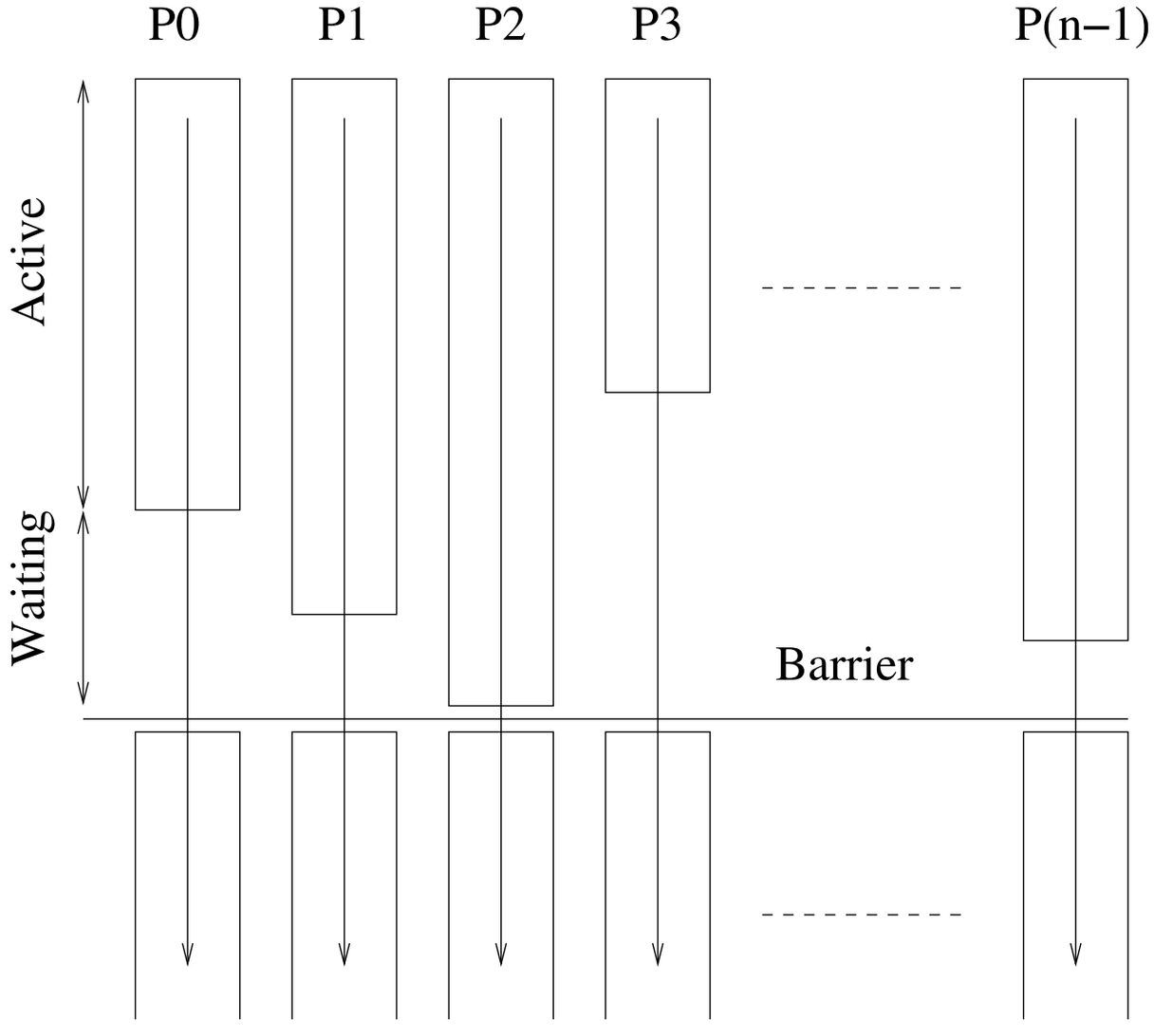
Bariera

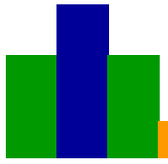
Bariera:

- bariera este *mecanismul de bază* pentru sincronizarea proceselor
- o *instrucţiune* de barieră trebuie inserată *în fiecare proces* în punctul unde trebuie aşteptat pentru sincronizare
- un proces poate *continua* din punctul de sincronizare când *toate procesele au ajuns la bariera* (ori, în cazul unor sincronizări parţiale, când un număr dat de procese au ajuns la barieră).

..Bariera

Procese cae ajungând la barieră la *timpi diferiţi*:

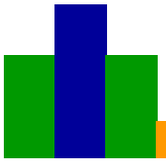




Bariera in MPI

In sistemele cu message-passing, barierele sunt adesea *rutine* de bibliotecă

- MPI: `MPI_Barrier()`
  - este o barieră cu un unic argument, anume numele *grupul comunicator* care trebuie sincronizat
  - trebuie invocată de *fiecare proces* din grup, blocând procesele până ce toţi membrii grupului au atins bariera



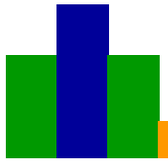
Iterare sincronă

Iterare sincronă: termenul este folosit pentru a descrie o situaţie în care o problemă este *rezolvată prin iterare* şi:

- *fiecare pas de iterare* este compus din diverse procese care *încep în acelaşi timp* pasul de iterare şi
- *următorul pas de iterare* nu poate începe pâna când *toate procesele au sfârşit* pasul de iterare curent.

Exemplu:

```
for (j=0; j<n; j++){
    i = myrank;
    body(i);
    barrier(mygroup);
}
```



Exemplu: Problema distributiei caldurii

Problema distributiei caldurii: Presupunem dată *o bucată de metal* la care se ştie *temperatura pe margini*. Scopul este de a afla *distribuţia temperaturii în metal* (în starea de echilibru).

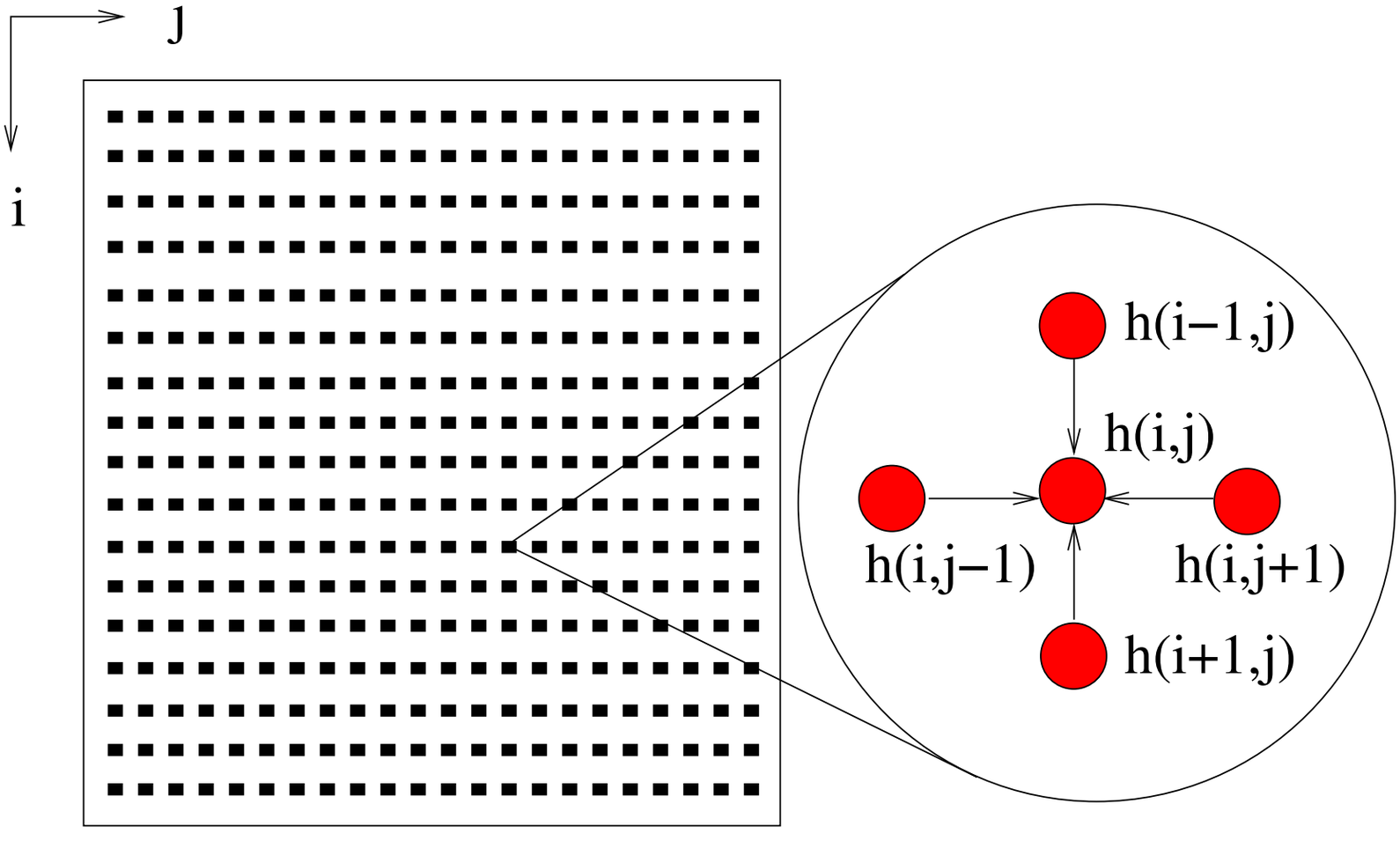
Soluţia:

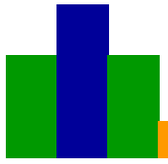
- împărţim aria într-o reţea fină de puncte $h_{i,j}$
- evoluţia în timp este dată de următoarea relaţie

$$h_{i,j}^{t+1} = \frac{h_{i-1,j}^{t} + h_{i+1,j}^{t} + h_{i,j-1}^{t} + h_{i,j+1}^{t}}{4}$$

- cum se întâmplă în metodele iterative, *repetăm* un număr prestabilit de iterate, ori până ce diferenţa dintre două iterate consecutine devine în fiecare punct mai mică decât un prag dat.

..Distributia caldurii





..Distributia caldurii

Legătura cu sistemele liniare: Pentru simplitate, notăm punctele de la 1 la k^2. Fiecare punct are asociată o ecuaţie

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-k} + x_{i+k}}{4}$$

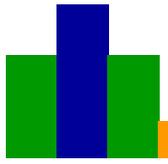
rezultând un sistem de ecuaţii liniare

$$x_{i-k} + x_{i-1} - 4x_i + x_{i+1} + x_{i+k} = 0$$

Acesta este sistemul cu *diferenţe finite* asociat *ecuaţiei Laplace*.

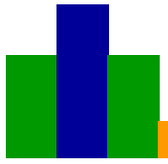
Inapoi la indici dublii: Temperatura pe margini este ştiută, deci ştim $h_{0,r}, h_{n,r}, h_{r,0}, h_{r,n}$ pentru orice $0 \leq r \leq n$.

Un cod secvenţial (ce include o condiţie de terminare) este:



..Distributia caldurii

```
do{
   for (i=1; i<n; i++)
      for (j=1; j<n; j++)
         g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]
                         +h[i][j-1]+h[i][j+1]);
   for (i=1; i<n; i++)
      for (j=1; j<n; j++)
         h[i][j] = g[i][j];
   cont = FALSE;
   for (i=1; i<n; i++)
      for (j=1; j<n; j++)
         if (!converged(i,j){
            continue = TRUE;
            break;
         }
      }
} while (continue == TRUE)
```

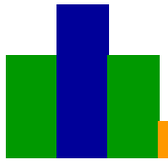


..Distributia caldurii

Cod paralel: (versiune cu un număr fix de iterate şi un proces $P_{i,j}$ pentru fiecare punct:)

```
for (iter=0; iter<limit; iter++)
    h[i][j] = 0.25*(w+e+n+s);
    send(&h[i][j], P_{i-1,j}); /* send ne-blocante */
    send(&h[i][j], P_{i+1,j});
    send(&h[i][j], P_{i,j-1});
    send(&h[i][j], P_{i,j+1});
    recv(&w, P_{i-1,j}); /* receive blocant */
    recv(&e, P_{i+1,j});
    recv(&n, P_{i,j-1});
    recv(&s, P_{i,j+1});
}
```

Notă: Este improtant să folosim send ne-blocant*, altfel procesele se blochează. Pe de altă parte, trebuie ceva blocant (ca* receive-ul blocant *folosit) pentru sincronizare.*



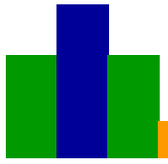
..Distributia caldurii

Trebuie atenţie pentru *procesele care se află pe margine*:

- se pot aloca procese pe margine care *doar trimit datele*; e.g., pentru $P_{0,j}$

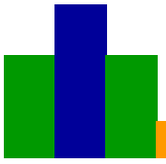
```
for (iter=0; iter<limit; iter++)
    send(&h[0][j],P1,j)
```

- altă posibilitate este de a *şterge* instrucţiunile `send/recv` din codul proceselor care se află lângă margine, e.g.,



..Distributia caldurii

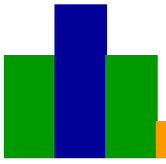
```
  ⋮
if (i != 1) send(&h[i][j],P_{i-1,j});
if (i != n-1) send(&h[i][j], P_{i+1,j});
if (j != 1) send(&h[i][j], P_{i,j-1});
if (j != n-1) send(&h[i][j], P_{i,j+1});
if (i != 1) recv(&w, P_{i-1,j});
if (i != n-1) recv(&e, P_{i+1,j});
if (j != 1) recv(&n, P_{i,j-1});
if (j != n-1) recv(&s, P_{i,j+1});
⋮
```



..Distributia caldurii

Partiţionare: Cum numărul de puncte este uzual mare, trebuie alocate mai multe puncte aceluiaşi proces. Partiţionări naturale sunt în *blocuri pătrate* ori *benzi*.

- Parţiţie în blocuri: Timpul de comunicare este $t_{commsq} = 8(t_{startup} + \sqrt{(n/p)}t_{data})$
- Parţiţie în benzi: Timpul de comunicare este $t_{commcol} = 4(t_{startup} + \sqrt{n}t_{data})$.



..Distributia caldurii

Timpul de comunicare: Benzile sunt mai bune când timpul de stabilire a comunicării (startup time) este mare; pentru timp startup mic, blocurile sunt mai bune. Intr-adevăr:

benzile sunt mai bune decât blocurile

dnd

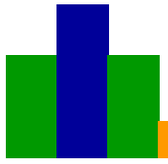
$$t_{commcol} < t_{commsq}$$

dnd

$$4(t_{startup} + \sqrt{n}t_{data}) < 8(t_{startup} + \sqrt{(n/p)}t_{data})$$

dnd

$$t_{startup} > \sqrt{n}(1 - \frac{2}{\sqrt{p}})t_{data}$$

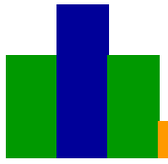


..Distributia caldurii

Detalii de implementare:

- o tehnică bună este de a adăuga o linie adiţională de puncte (*puncte fantomă*) la marginea ariei unui proces care să memoreze valorile proceselor învecinate
- cod cu puncte fantomă (pe linii; $m = \sqrt{n}$ şi $m1 = m/p$)

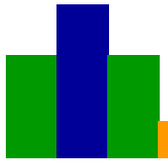
```
for (i=1; i<m1; i++)
     for (j=1; j<m; j++)
          g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]
                         +h[i][j-1]+h[i][j+1]);
for (i=1; i<m1; i++)
     for (j=1; j<m; j++)
          h[i][j] = g[i][j];
send(&g[1][1],&m, P_{i-1});
send(&g[1][m],&m, P_{i+1});
recv(&h[1][0],&m, P_{i-1});
recv(&h[1][m+1],&m, P_{i+1});
```



..Distributia caldurii

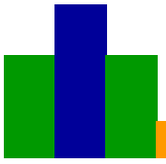
Send/receive nesigur:

- Dacă *toate* procesele *întâi trimit, apoi primesc* trebuie un spaţiu mare pentru buffer. Dacă *nu există spaţiu suficient*, atunci rutinele `send()` devin local blocante şi *poate apare deadlock*.
- O *soluţie* este de a alterna rutinele `send/recv`. E.g., dacă avem partiţii pe linii putem folosi:



..Distributia caldurii

```
if(myid % 2) == 0){
     send(&g[1][1],&m,P_{i-1});
     recv(&h[1][0],&m,P_{i-1});
     send(&g[1][m],&m,P_{i+1});
     recv(&h[1][m+1],&m,P_{i+1});
} else {
     recv(&h[1][0],&m,P_{i-1});
     send(&g[1][1],&m,P_{i-1});
     recv(&h[1][m+1],&m,P_{i+1});
     send(&g[1][m],&m,P_{i+1});
}
```

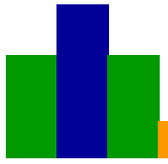


..Distributia caldurii

Alternative pentru o comunicare sigură:

- se *combină send şi receive* în aceeaşi rutină `MPI_Sendrecv()` (care garantat nu se blochează)
- se foloseşte `MPI_Bsend()`, unde utilizatorul asigură *spaţiu explicit pentru buffer*
- se folosesc rutine neblocante ca `MPI_Isend()` şi `MPI_Irecv()` - *rutinele se termină imediat* şi se folosesc rutine separate pentru a testa dacă comunicarea s-a terminat cu succes

Notă: exemple de rutine MPI pentru a testa succesul comunicării:
`MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`, `MPI_Testall()`, `MPI_Testany()`.



Sisteme distribuite

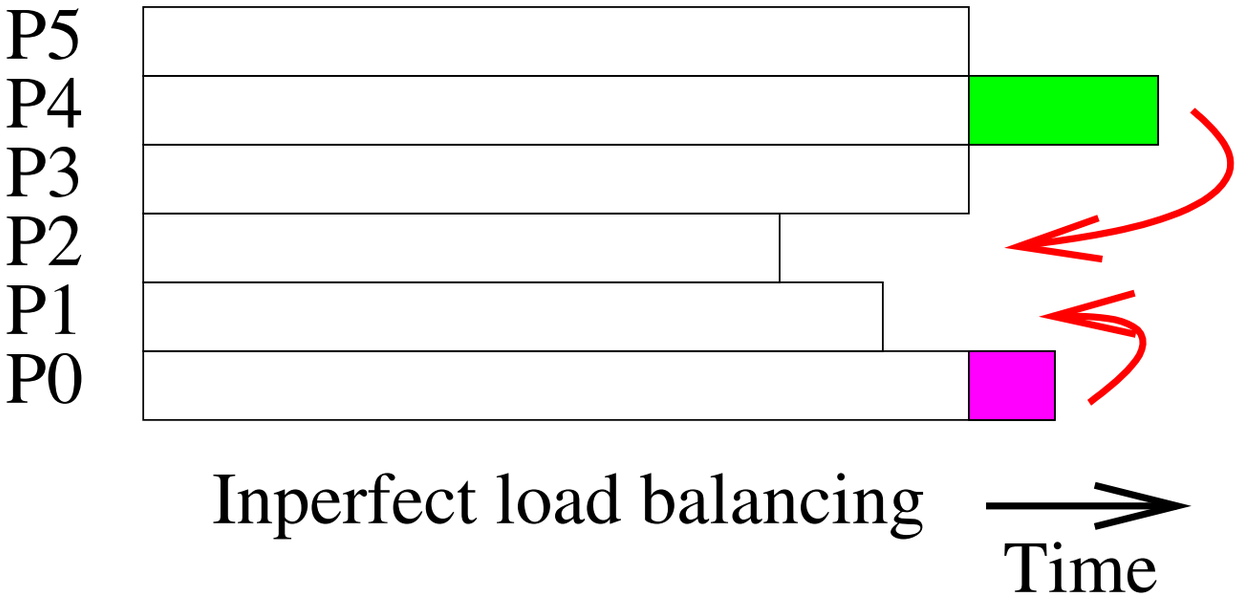
Balansarea execuţiei (load balancing) este:

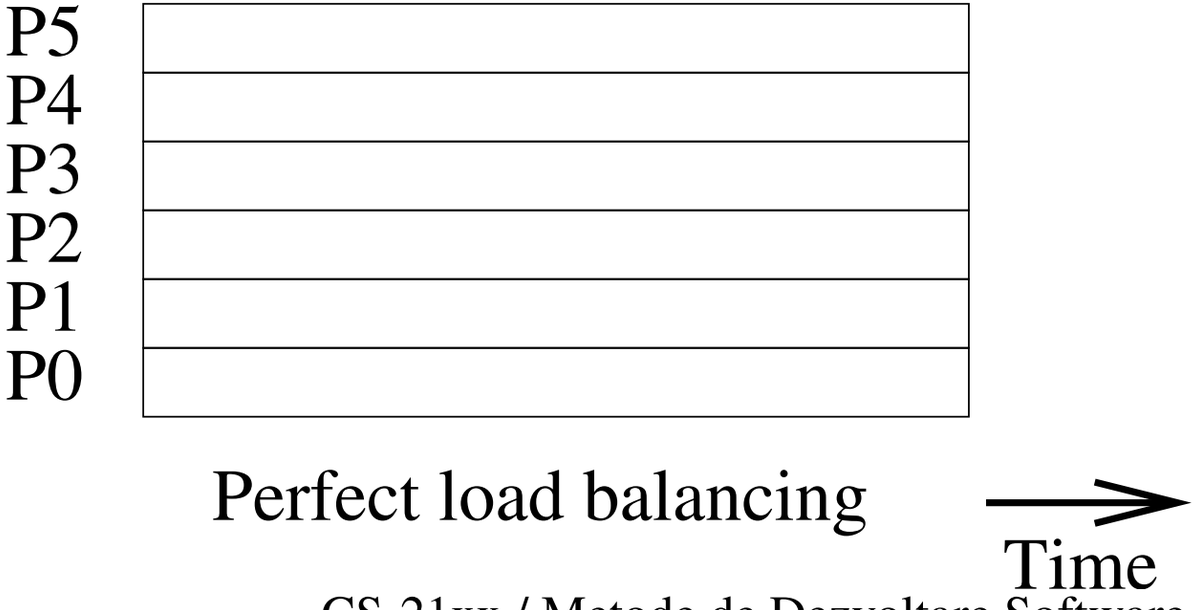
- o tehnică care face o *distribuţie onestă a calculului* pe procese spre a *creşte viteza de calcul*

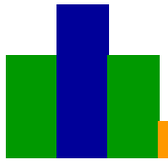
Detecţia terminarii:

- se detectează când calculul s-a terminat
- este de regulă *dificilă* când procesele sunt *distribuite* (un proces terminat poate fi ulterior reactivat)

Load balancing





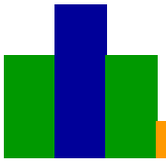


Balansare statica (SLB)

Balansarea statică: în acest caz planificarea balansării trebuie făcută *înainte* de execuţia proceselor.

Exemple de tehnici de balansare statică:

- *Round robin algorithm*: job-urile sunt pasate proceselor în ordine secvenţială; când ultimul proces a primit un job, planificarea continuă cu primul proces (o nouă rundă)
- *Randomized algorithm*: alocarea job-urilor proceselor se face aleator
- *Recursive bisection*: se divide problema recursiv în subprobleme cu efort calculatoriu aproximativ egal
- *Simulated annealing* ori *genetic algorithms*: mixturi de alocări care folosesc tehnici de optimizare



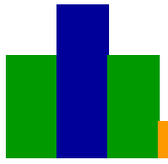
Un rezultat teoretic

Un rezultat teoretic:

- Problema *alocării de job-uri la procese într-o reţea arbitrară* este *NP-hard*

Deci

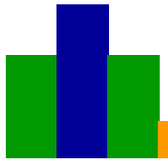
- Cum în genere se crede că $P \neq NP$, *nu există algoritmi polinomiali* şi trebuie folosite diverse *heuristici*



Critici la balansarea statica

Critici: - chiar dacă ar exista soluţii matematice bune, balansarea statică are încă multe neajunsuri:

- este dificil de *estimat a-priori* [cu acurateţe] *timpul de execuţie* al diverselor părţi din program
- uneori există *întârzieri de comunicare* care pot varia incontrolabil
- pentru unele probleme, *numărul de paşi* pentru a ajunge la soluţie *nu se ştie* în avans



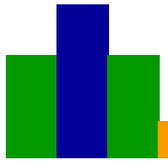
Balansare dinamica (DLB)

Balansarea dinamica:

- Planificarea alocării de job-uri este făcută *în timpul execuţiei* proceselor

Caracteristici:

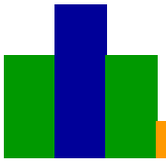
- criticile de mai sus dispar
- există o *încărcare adiţională* în timpul execuţiei, dar, de regulă, este mai eficientă decât balansarea statică
- *detecţieq terminării* este *mai complicată* cu balansare dinamica



..DLB

(Caracteristici, cont.)

- calculul este divizat în *job-uri* ori *sarcini* ce trebuie procesate, iar *procesele* execută aceste sarcini
- procesele se alocă *procesoarelor*; în final, scopul este de a ţine procesoarele ocupate;
- deseori, se alocă un singur proces pe procesor, deci terminii *proces* şi *procesor* sunt cumva interschimbabili.



Tipuri de DLB

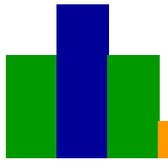
Balansarea dinamică poate fi clasificată ca *centralizată* ori *descentralizată*.

Balansarea dinamică centralizată:

- job-urile se alocă proceselor dintr-o locaţie centralizată
- tipic, structura de comunicare este *master-slave*

Balansarea dinamică descentralizată:

- procesele (lucrătoare) interacţionează între ele spre a rezolva problema, în final raportând unui singur proces
- job-urile sunt create şi pasate între procese arbitrare: un proces lucrător poate recepţiona job-uri de la oricare alt proces şi poate trimite job-uri la oricare alt proces



DLB centralizat

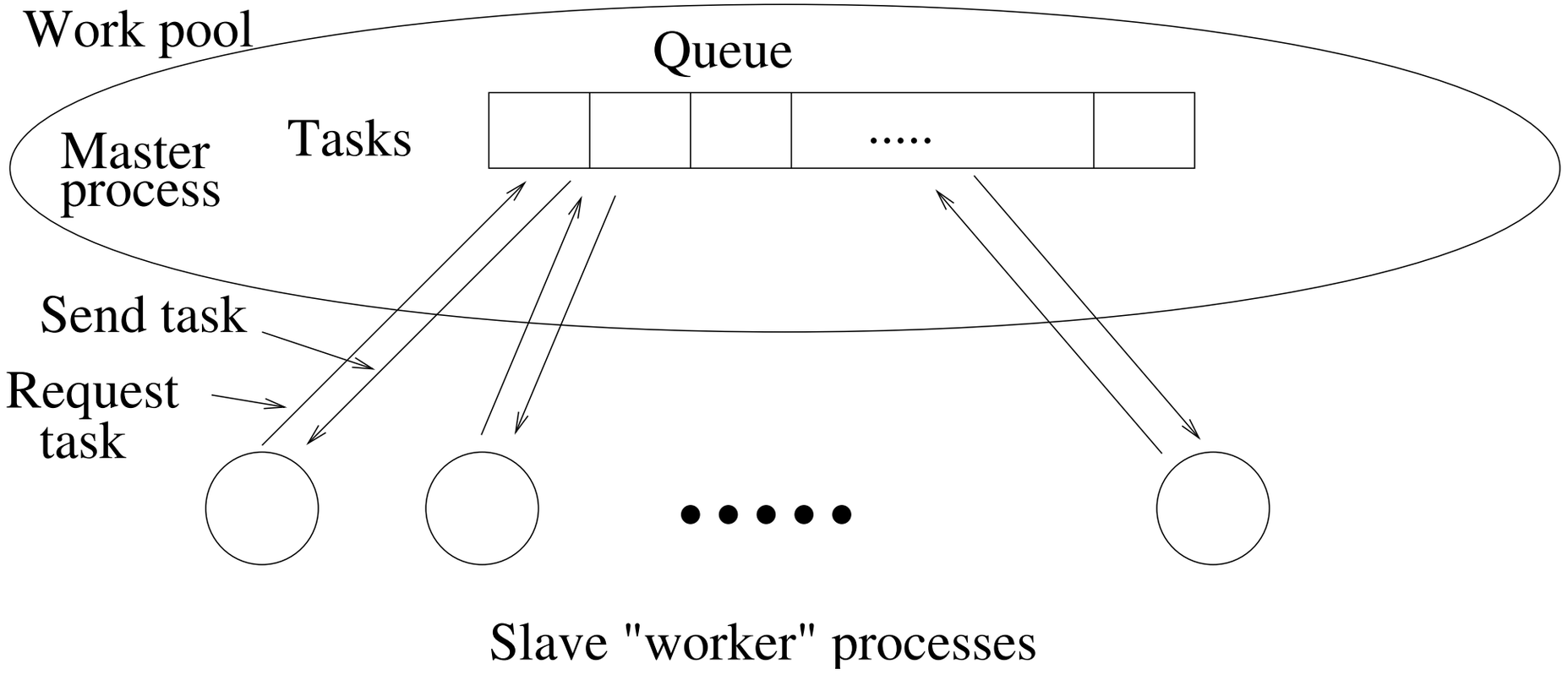
Este bun când: există un *număr mic de sclavi* şi problema conţine job-uri *cu calcul intensiv*

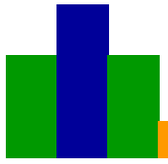
Caracteristici de bază:

- un proces master deţine o colecţie de job-uri de procesat
- job-urile sunt trimise proceselor sclav
- când un proces sclav termină un job cere un nou job de la procesul master

Deseori se folosesc termenii de work pool*, ori* replicated worker*, ori* processor farm *pentru acest caz. Tehnic, este mai eficient să se înceapă cu job-urile mai dificile.*

..DLB centralizat





Terminarea în DLB centralizat

Terminare: Există diverse cazuri

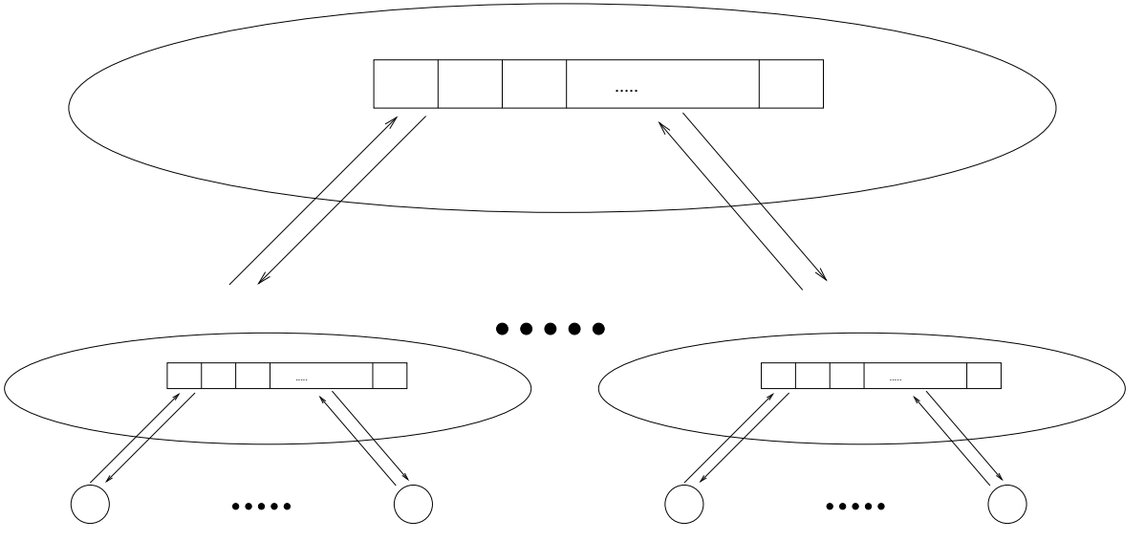
1. Calculul se opreşte dacă soluţia a fost găsită
2. Dacă job-urile sunt luate dintr-o coadă de job-uri, calculul se termină când
   - coada de job-uri este vidă *şi*
   - orice proces a cerut un job fără ca noi job-uri să mai fie generate

   [Nu este suficient să se golească coada, căci pot exista procese care nu s-au terminat şi pot pune noi job-uri în coadă.]
3. In unele aplicaţii, un proces sclav poate detecta singur terminarea printr-un criteriu local, spre exemplu în algorithmi de căutare

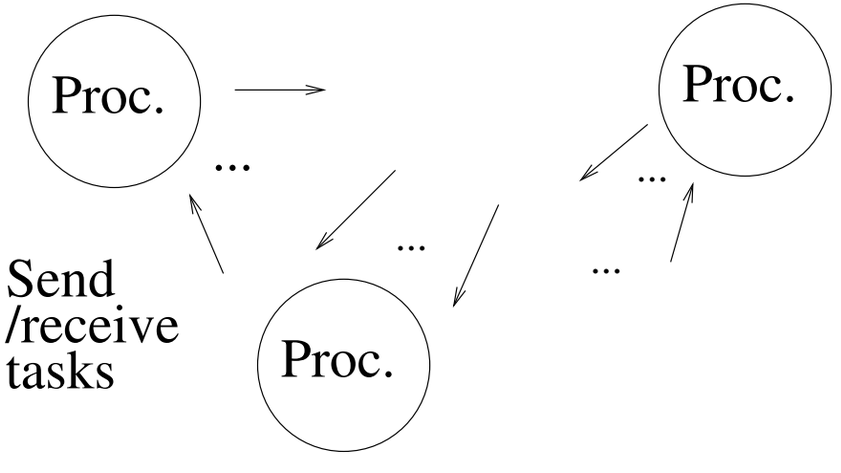
DLB descentralizat

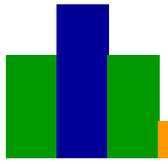
Work pool distribut

Strucutră arborescenă:



General (work pool complet distribuit):





Mecanisme de transfer de job-uri

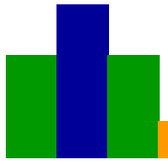
Transfer de job-uri: prin

(1) *metodă iniţiată de cel ce primeşte* ori

(2) *metodă iniţiată de cel ce trimite*

Metodă iniţiată de cel ce primeşte:

- *un proces cere job-uri* de la alte procese alese de el; de regulă, face asta, când are puţine job-uri de procesat, ori nu are de loc
- metoda este utilă când sistemul are *o încărcătură mare* [cum am mai spus, este greu de detectat apriori încărcătura unui sistem]



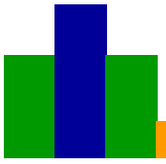
..Transfer de job-uri

Metodă iniţiată de cel ce trimite:

- *un proces trimite job-uri* la alte procese alese de el; de regulă, face asta când are o încărcătură mare şi poate pasa unele job-uri altor procese care sunt doritoare să le primească
- metoda este utilă când *sistemul nu este prea încărcat*

Comentarii finale:

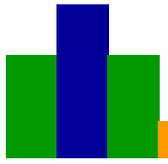
- abordările “pure” de mai sus pot fi mixate
- totuşi, orice metodă s-ar aplica, echilibrarea încărcării pe procese este greu de realizat dacă lipsesc procese disponibile.



Selecţia proceselor în DLB

Selecţia proceselor: Algoritmi posibili de selecţie a proceselor

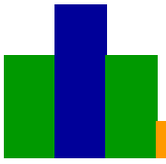
- *Round robin algorithm:* procesul P_i cere job-uri, pe rând, de la procesul P_x, unde x este un contor care se incrementează modulo n, [dacă sunt n procese], excluzând $x = i$
- *Random polling algorithm:* procesul P_i cere job-uri de la procesul P_x, unde x este un număr selectat aleator din mulţimea $\{0,\ldots,i-1,i+1,\ldots,n-1\}$ [presupunând că sunt n procese $P_0,\ldots P_{n-1}$]



DLB pe o structură lineară

Procedură: (informală)

- procesul *master alimentează linia cu job-uri* la un capăt; job-urile se shiftează pe linie
- *când un proces sclav* $P_i (1 \leq i < n)$ *detectează un job* şi este *inactiv*, *ia job-ul* de pe linie
- apoi *job-urile* se shiftează la dreapta spre a se umple locul lăsat liber de job, iar procesul master *inserează un nou job* pe linie
- scopul este ca toate procesele să aibă un job şi linia să fie plină cu job-uri
- pentru eficienţă, este mai bine ca joburi-le cu prioritate mare ori necesitând mai mult efort să fie puse primele pe linie



Cod (DLB pe o structură lineară)

Acţiuni de shiftare: - bazate pe comunicare la stânga şi la dreapta cu procesele adiacente şi pe execuţia job-ului curent

Cod (master):

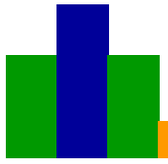
```
for (i=0; i < noTasks; i++){
    recv(P1, requestTag);
    send(&task,P1,taskTag);
}
recv(P1,requestTag);
send(&empty,P1,taskTag);
```

..Cod (DLB pe o structură lineară)

Cod $P_i (1 \leq i < n)$*:*

```
if (buffer == empty){
    send(P_{i-1},requestTag);
    recv(&buffer,P_{i-1},taskTag);
}
if ((buffer == full) && (!busy)){
    task = buffer;
    buffer = empty;
    busy = TRUE;
}
nrecv(P_{i+1}, requestTag, request)
if (request & (buffer == full)){
    send(&buffer, P_{i+1});
    buffer = empty;
}
if (busy) {
    do some work on task;
    if task finished, set busy to false;
}
```

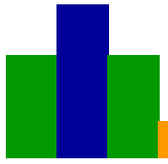
Notă: `nrecv` *este rutina receive neblocantă.*



Rutine neblocante

MPI:

- Rutina receive neblocantă, `MPI_Irecv()`, returnează într-un parametru un request "handle", care este folosit ulterior pentru a completa comunicarea (folosind `MPI_Wait` şi `MPI_Test`)
- de fapt, ea postează o cerere pentru un mesaj şi returnează imediat

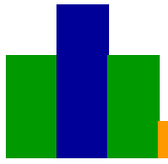


Detecţia terminării (cazul distribuit)

Condiţii de terminare: - trebuie satisfăcute următoarele condiţii de terminare:

- *condiţiile locale* de terminare sunt satisfăcute de toate procesele
- nu există mesaje *în tranzit* între procese

Notă: A doua condiţie este necesară spre a evita situaţia când un mesaj în tranzit poate restarta *un proces deja terminat. Nu este uşor de verficat căci timpul de comunicare nu poate fi ştiut în avans.*

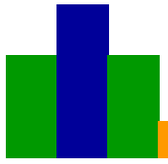


Dual-pass ring termination algorithm

Dual-pass ring termination algorithm:

- poate rezolva cazurile când *procesele pot fi reactivate după terminarea lor locală*, dar necesită trecerea repetată a unui jeton prin ring.
- tehnic, se folosesc *jetoane colorate*: un jeton poate fi *negru* ori *alb*
- ca o consecinţă, *procesele* de asemenea sunt *colorate*: un proces este ori *negru* ori *alb*
- în esenţă, culoarea neagră înseamnă că condiţiile de terminare globală nu au fost atinse

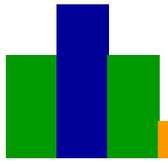
Algoritmul este următorul (începând cu P_0):



..dual-pass ring termination

(..dual-pass ring termination algorithm):

- P_0 devine alb când se termină; el generează un jeton alb şi îl pasează lui P_1
- Jetonul este pasat prin ring de la un proces P_i la următorul când P_i s-a terminat.
- Dar, culoarea jetonului se poate schimba: dacă un proces P_i pasează un job unui proces P_j cu $j < i$, atunci el devine un *proces negru*; altfel el este/rămâne *proces alb*.
- Un proces negru colorează jetonul negru şi il trimite mai departe; un proces alb trimite jetonul cum l-a primit
- După ce P_i a pasat jetonul devine un proces alb.
- Dacă P_0 primeşte un jeton negru, pasează un jeton alb; dacă primeşte un jeton alb, toate procesele s-au terminat.

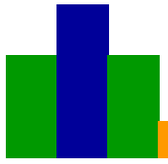


Exemplu: Cel mai scrut drum (SPP)

Cel mai scurt drum: - problema este de a găsi cea mai scurtă distanţă dintre două puncte ale unui graf; mai precis,

SPP Dată o mulţime de noduri interconectate, unde arcele dintre noduri sunt marcate cu *ponderi*, să se găsească un drum între două noduri specificate care are cea mai mică pondere acumulată.

Convenţii: se foloseşte un *graf* format din *vârfuri* şi *arce*; dacă arcele au direcţie [i.e., pot fi traversate doar într-o direcţie] avem un graf *direcţonat*.



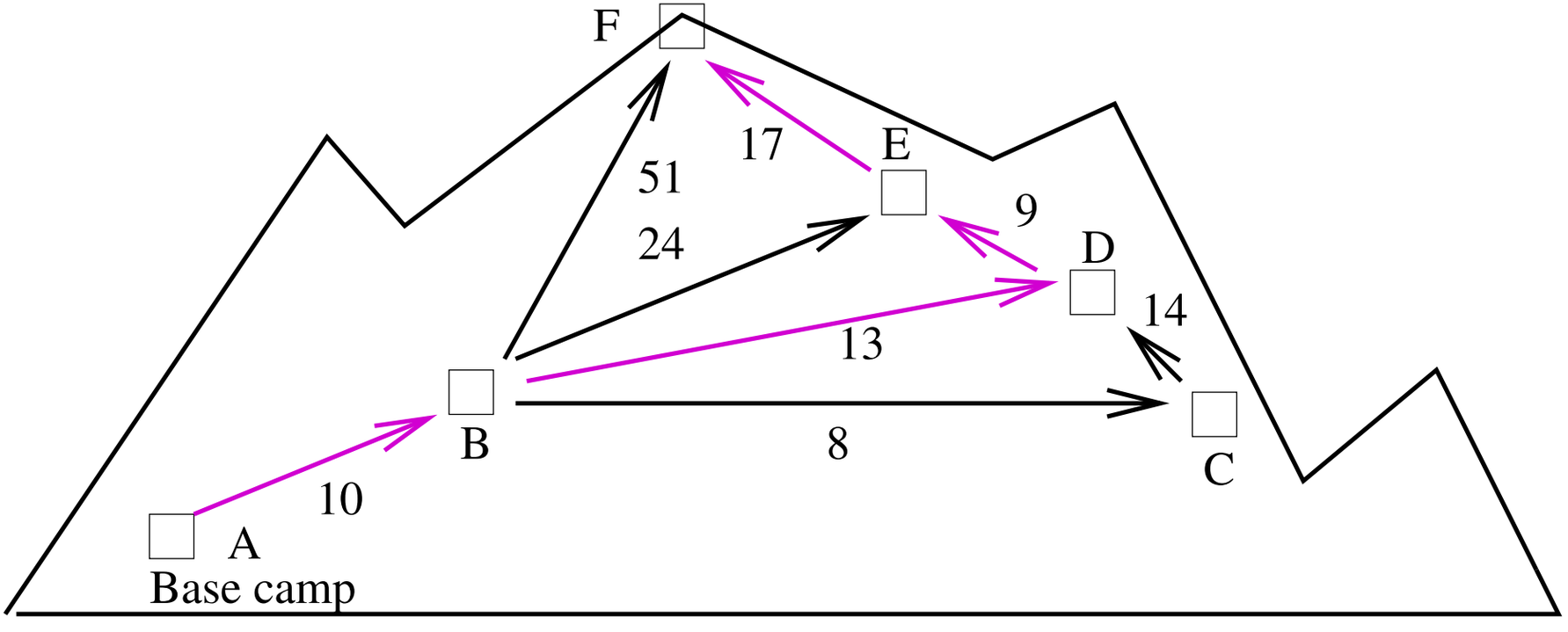
Interpretari particulare pentru SPP

Interpretari particulare:

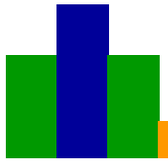
- A afla cea mai mică distanţă dintre două puncte de pe o hartă, unde ponderile sunt distanţele;
- A afla cea mai rapidă rută de calătorie, unde ponderile reprezintă timpul [diferit de “distanţă”, dacă se foloseşte avion, tren, etc.];
- Cel mai ieftin mod de călătorie între două puncte;
- Cel mai bun mod de a cuceri un vârf de pe munte;
- Cea mai bună rută de comunicare într-o reţea (delay minim)
- Etc.

Cel mai bun mod de a cuceri un vârf

Exemplu:



- Ponderile indică gradul de efort între 2 puncte de pe hartă
- Efortul într-o direcţie este diferit de efortul în direcţia contrară, deci avem un graf *direcţionat*



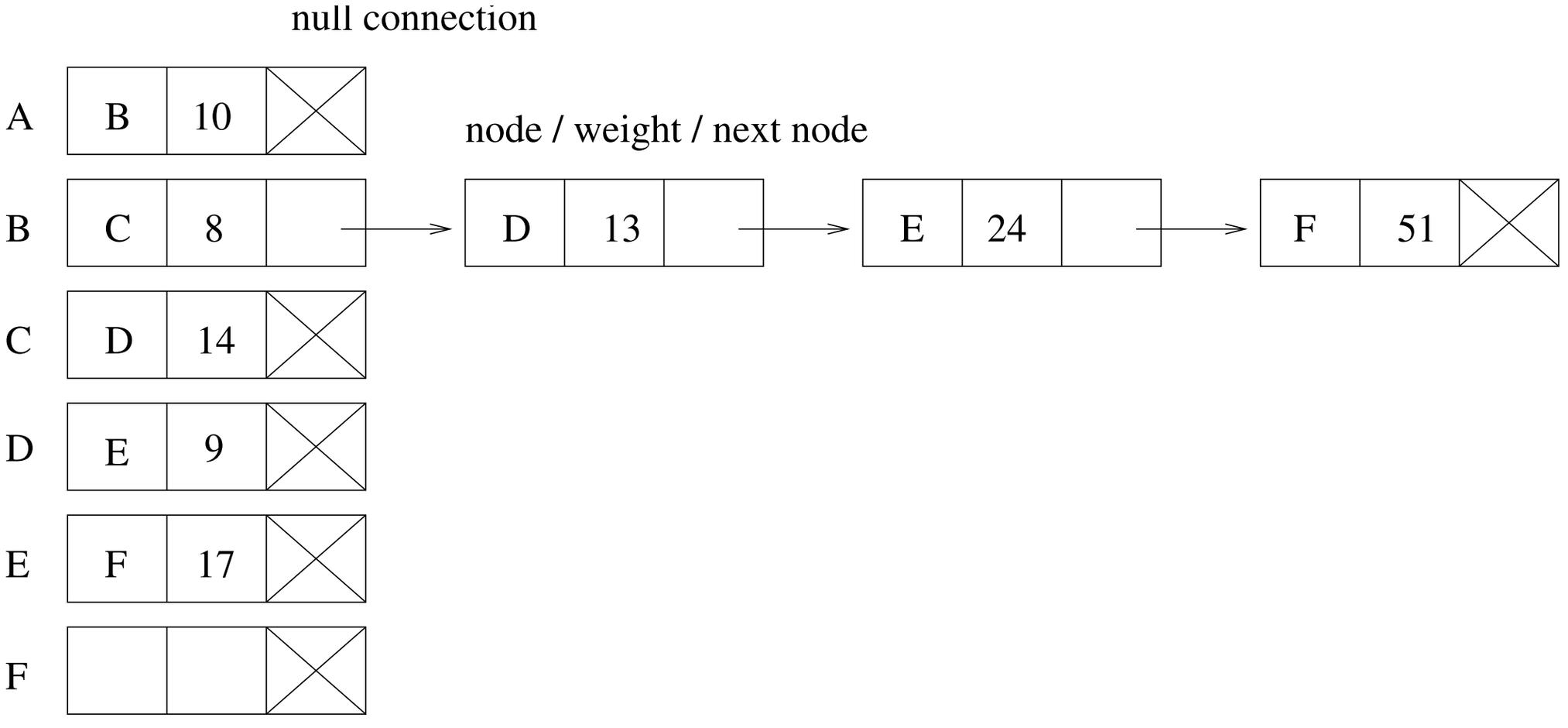
Reprezentari de grafuri

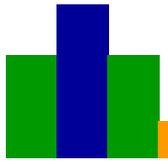
Matrici: - se foloseşte un tabel 2-dimensional `a`, în care `a[i][j]` reţine ponderea asociată arcului de la vârful *i* la vârful *j* [reprezentare bună pentru grafuri *dense*]

Source \ *Destination*	*A*	*B*	*C*	*D*	*E*	*F*
A	∞	10	∞	∞	∞	∞
B	∞	∞	8	13	24	51
C	∞	∞	∞	14	∞	∞
D	∞	∞	∞	∞	9	∞
E	∞	∞	∞	∞	∞	17
F	∞	∞	∞	∞	∞	∞

..Reprezentari de grafuri

Liste: unui vârf *v* îi ataşăm o listă conţinând toate vârfurile conectate direct cu *v* (printr-un arc) şi reţinem ponderea arcului [bună pentru grafuri *rare/sparse*]





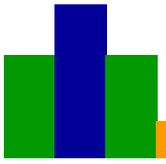
Algorithmi secvenţiali pentru SPP

Doi algorithmi bine-cumoscuţi (algorithmi "single-source shortest path")

- Algorithmul lui Moore (1957); se foloseşte orice ordine de selecţie
- Algorithmul lui Dijkstra (1959); se începe cu cel mai apropiat vârf

Noi alegem algorithmul lui Moore căci este mai uşor de paralelizat, deşi lucrează mai mult.

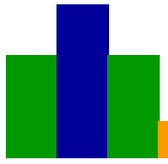
[Notă: Ponderile trebuie să fie valori *pozitive*; există variaţii care rezolvă şi cazurile cu numere negative.]



Algorithmul Moore

Algorithmul Moore: - bazat pe $d_j := min(d_j, d_i + w_{i,j})$

- Se pleacă cu vârful sursă;
- Fie d_i distanţa minimă [actualizată] de la sursă la nodul i; pentru un vârf j arbitrar, actualizăm distanţa minimă d_j cu $d_i + w_{i,j}$, dacă cea din urmă este mai mică;
- Repetăm cele de mai sus pentru toate vârfurile cu distanţa actualizată, până nu mai sunt modificări neprocesate.



..Algorithmul Moore

Code secvenţial:

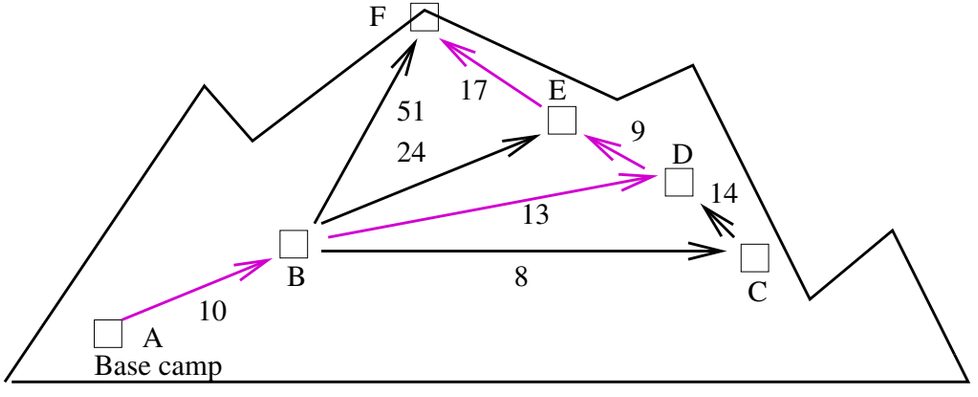
```
while(vertexQ != empty){
    i = getVetex(vertexQ);
    for (j=1; j<n; j++){
        if (w[i][j] != infinity){
            newDist = dist[i]+w[i][j];
            if(newDist < dist[j]){
                dist[j] = newDist;
                put(j,vertexQ);
            }
        }
    }
}
```

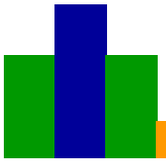
..Algorithmul Moore

Exemplu:

Folosim o coadă FIFO `vertexQ` care reţine toate vârfurile cu distană actualizată; `dist[i]` reţine distanţa curentă la vârful i. O execuţie, pe exemplu precedent, este

`vertexQ`	`dist` *to*					
	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
B	0	10	∞	∞	∞	∞
$E, D, C[, F]$	0	10	18	23	34	61
D, C	0	10	18	23	34	51
C, E	0	10	18	23	32	50
$E[, D]$	0	10	18	23	32	49
$[, F]$						





Algorithmul Moore paralel, centralizat

Explicaţii:

- vârfurile din `vertexQ` se folosesc drept job-uri
- fiecare proces sclav ia vârfuri din coadă şi returneaxă noi vârfuri în coadă
- procesul master reţine un vector cu distanţele curente; el actualizează vectorul dacă primeşte distanţe mai scrute
- cum graful este fix, presupunem că toate procesele sclav au a copie a matricii de adiacenţă

..Algorithmul Moore paralel, centralizat

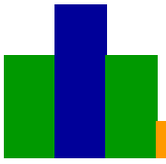
Cod master

```
while ((vertexQ != empty) || (more messages to come)){
     recv(j, newDist, P_any, source = P_i, tag);
     if (tag == requestTag){                          /* request task */
          v = get(vertexQ);
          send(v,P_i);                              /* send next vertex */
          send(&dist,&n,P_i);                         /* and dist array */
     } else {                                 /* got vertex and distance */
          if(newDist[j] < dist[j]){
               put(j,vertexQ);                  /* put vertex in queue */
               dist[j] = newDist;                        /* update dist */
          }
     }
}
for (j=1; j<n; j++){
     recv(P_any, source = P_i);
     send(P_i, terminationTag);
}
```

..Algorithmul Moore paralel, centralizat

Cod sclav

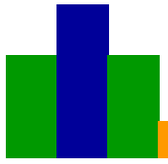
```
send(P_master)                                  /* send request for task */
recv(&v,P_master, tag)                          /* get vertex/tag */
while (tag != termination tag){
     recv(&dist,&n,P_master);                   /* get distances */
     for (j=1; j<n; j++)
          if(w[v][j] != infinity){
               newDist = dist[v]+w[v][j]
               if (newDist < dist[j]){
                    send(&j,&newDist,P_master);
                                   /* send vertex and new distance */
               }
          }
     send(P_master)                             /* send request for task */
     recv(&v,P_master, tag)                     /* get vertex/tag */
}
```



Algorithmul Moore paralel, descentralizat

Prezentare informală:

- Procesul i este alocat vârfului i; el reţine distanţa minimă [curentă] `dist` de la vârful sursă la nodul i [iniţial aceasta este ∞]; de asemenea, reţine o listă cu vârfurile vecine
- Când un proces i primeşte o nouă distanţă, mai mică, de la sursă la el însuşi, atunci
  —actualizează distanţa sa minimă curentă `dist`
  —calculează toate noile distanţe `dist+w[j]` la toţi vecinii săi j, şi
  —trimite aceste valori vecinilor
- De regulă procesele sunt inactive; ele sunt activate când primesc distanţe de la alte procese;
- Terminarea se detectează cu un algorithm de detectie a terminării, ca în orice sistem distribuit.



..Algorithmul Moore paralel, descentralizat

Cod (sclav)

```
recv(newDist, P_any);
if (newDist < dist){
    dist = newDist;
    for (j=1; j<n; j++){
        if (w[j] != infinity){
            d = dist + w[j];
            send(&d, P_j);
        }
    }
}
```

Notă: Acesta este doar pasul de bază; el trebuie pus într-un loop, şi integrată procedura de terminare.