CPA – Computació Paral·lela

Grau en Enginyeria Informàtica

S3. Programació amb MPI

J. M. Alonso, P. Alonso, F. Alvarruiz, I. Blanquer,
J. Ibáñez, E. Ramos, J. E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curs 2024/25

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Contingut

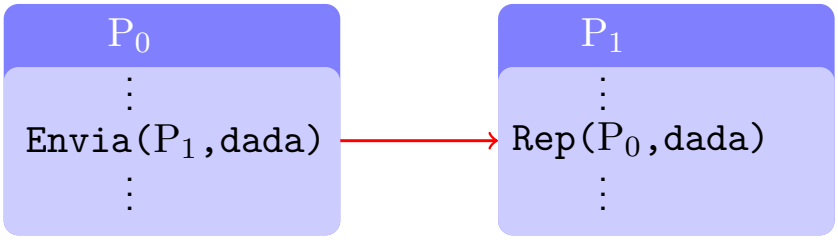
Apartat 1

Conceptes Bàsics

- Model de Pas de Missatges
- L'Estàndard MPI
- Model de Programació MPI

Model de Pas de Missatges

Intercanvi d'informació mitjançant enviament i recepció explícits de missatges



Model més usat en computació a gran escala – Biblioteques de funcions (aprenentatge més fàcil que un llenguatge nou)

Avantatges:

- Universalitat
- Fàcil comprensió
- Gran expressivitat
- Major eficiència

Inconvenients:

- Programació complexa
- Control total de les comunicacions

L'Estàndard MPI

MPI és una especificació proposada per un comitè d'investigadors, usuaris i empreses

`https://www.mpi-forum.org`

Especificacions:

- MPI-1.0 (1994), última actualització MPI-1.3 (2008)
- MPI-2.0 (1997), última actualització MPI-2.2 (2009)
- MPI-3.0 (2012), última actualització MPI-3.1 (2015)
- MPI-4.0 (2020), última actualització MPI-4.1 (2020)

Antecedents:

- Cada fabricant oferia el seu propi entorn (migració costosa)
- PVM (*Parallel Virtual Machine*) va ser un primer intent d'estandardització

Característiques de MPI

Característiques principals:

- És portable a qualsevol plataforma paral·lela
- És simple (amb tan sols 6 funcions es pot implementar qualsevol programa)
- És potent (més de 300 funcions)

L'estàndard especifica interfície per a C i Fortran

Hi ha moltes implementacions disponibles:

- Propietàries: IBM, Cray, SGI, ...
- MPICH (`www.mpich.org`)
- Open MPI (`www.open-mpi.org`)
- MVAPICH (`mvapich.cse.ohio-state.edu`)

Model de Programació

La programació en MPI es basa en funcions de biblioteca
Per al seu ús, es requereix una inicialització

Exemple

```
#include <mpi.h>
int main(int argc, char* argv[]) {
  int k;       /* rang del procés     */
  int p;       /* número de processos */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &k);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  printf("Soc el procés %d de %d\n", k, p);
  MPI_Finalize();
  return 0;
}
```

- És obligatori cridar a `MPI_Init` i `MPI_Finalize`
- Una vegada inicialitzat, es poden realitzar diferents operacions

Model de Programació – Operacions

Les operacions es poden agrupar en:

- Comunicació punt a punt
  Intercanvi d'informació entre parells de processos
- Comunicació col·lectiva
  Intercanvi d'informació entre conjunts de processos
- Gestió de dades
  Tipus de dades derivades (p.e. dades no contigües en memòria)
- Operacions d'alt nivell
  Grups, comunicadors, atributs, topologies
- Operacions avançades (MPI-2, MPI-3)
  Entrada-eixida, creació de processos, comunicació unilateral
- Utilitats
  Interacció amb l'entorn del sistema

La majoria operen sobre comunicadors

Model de Programació – Comunicadors

Un comunicador és una abstracció que engloba els següents conceptes:

- *Grup*: conjunt de processos
- *Context*: per a evitar interferències entre missatges diferents

Un comunicador agrupa a p processos

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

Cada procés té un identificador (rang), un nombre entre 0 i $p-1$

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


Model d'Execució

El model d'execució de MPI segueix un esquema de creació simultània de processos en llançar l'aplicació

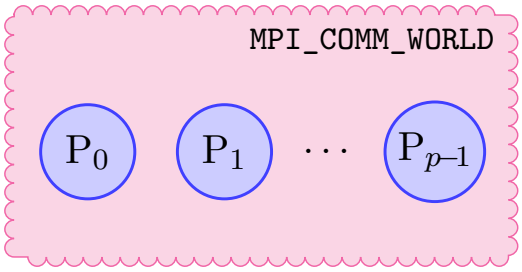
L'execució d'una aplicació sol fer-se amb

```
mpiexec -n p programa [arguments]
```

En executar una aplicació:

- Es llancen p còpies del mateix executable
- Es crea un comunicador `MPI_COMM_WORLD` que engloba a tots els processos

MPI-2 ofereix un mecanisme per a crear nous processos



Apartat 2

Comunicació Punt a Punt

- Semàntica
- Primitives Bloquejants
- Altres Primitives
- Exemples

Comunicació Punt a Punt – el Missatge

Els missatges han de ser enviats explícitament per l'emissor i rebuts explícitament pel receptor

Enviament estàndard:

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

Recepció estàndard:

```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

El contingut del missatge ve definit pels 3 primers arguments:

- Un *buffer* de memòria on està emmagatzemada la informació
- El nombre d'elements que componen el missatge
- El tipus de dades dels elements (p.e. `MPI_INT`)

Comunicació Punt a Punt – el Sobre

Per a efectuar la comunicació, és necessari indicar la destinació (`dest`) i l'origen (`src`)

- La comunicació està permesa només dins del mateix comunicador, `comm`
- L'origen i la destinació s'indiquen mitjançant identificadors de processos
- En la recepció es permet utilitzar `src=MPI_ANY_SOURCE`

Es pot utilitzar un nombre enter (etiqueta o `tag`) per a distingir missatges de diferent tipus

- En la recepció es permet utilitzar `tag=MPI_ANY_TAG`

En la recepció, l'estat (`stat`) conté informació:

- Procés emissor (`stat.MPI_SOURCE`), etiqueta (`stat.MPI_TAG`)
- Longitud del missatge (explicat en p. 43)

Nota: passar `MPI_STATUS_IGNORE` si no es requereix

Modes d'Enviament Punt a Punt

Existeixen els següents modes d'enviament:

- Mode d'enviament síncron
- Mode d'enviament amb memòria intermèdia (buffer)
- Mode d'enviament estàndard

El mode estàndard és el més utilitzat

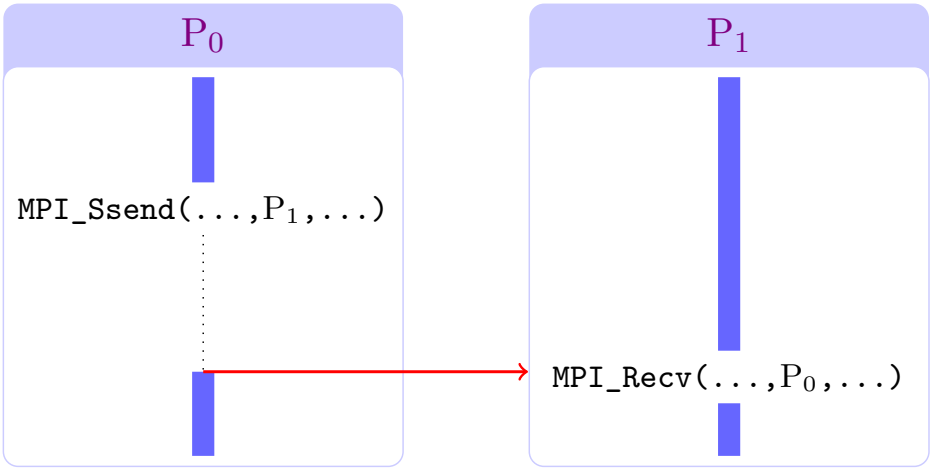
La resta de modes poden ser útils per a obtenir millors prestacions o major robustesa

Per a cada mode, existeixen primitives bloquejants i no bloquejants

Mode d'Enviament Síncron

```
MPI_Ssend(buf, count, datatype, dest, tag, comm)
```

Implementa el model d'enviament amb "rendezvous": l'emissor es bloqueja fins que el receptor desitja rebre el missatge

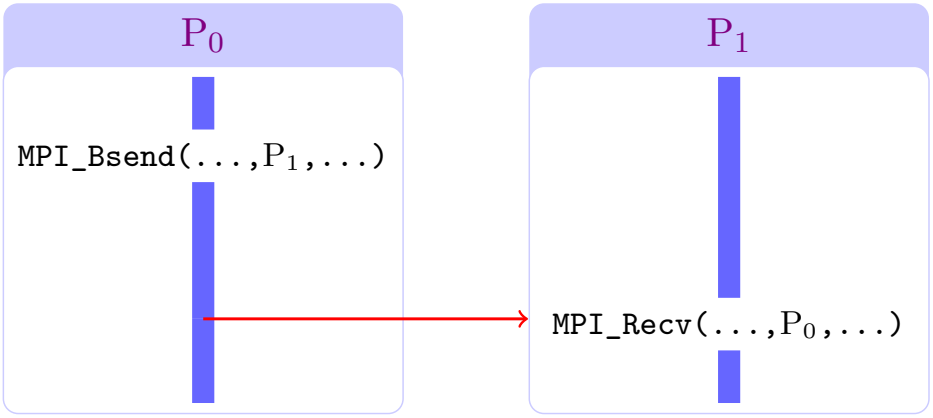


- Ineficient: l'emissor queda bloquejat sense fer res útil

Mode d'Enviament amb Buffer

```
MPI_Bsend(buf, count, datatype, dest, tag, comm)
```

El missatge es copia a una memòria intermèdia i el procés emissor continua la seua execució



- Inconvenients: còpia addicional i possibilitat de fallada
- Es pot proporcionar un buffer (`MPI_Buffer_attach`)

Mode d'Enviament Estàndard

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

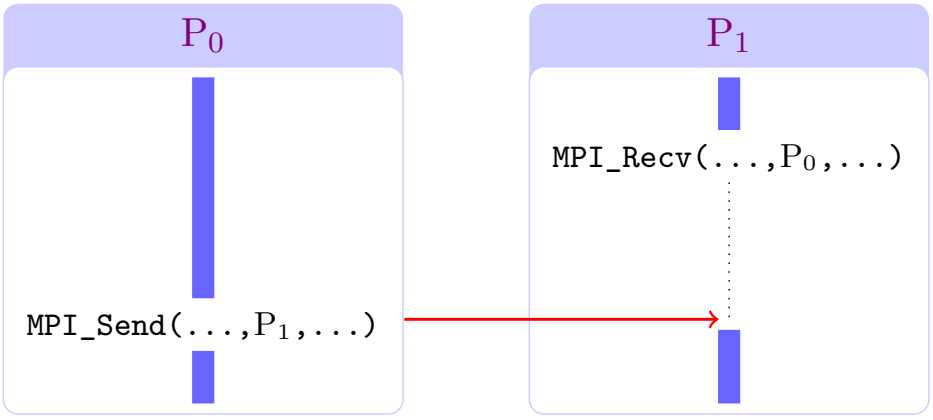
Garanteix el funcionament en tot tipus de sistemes ja que evita problemes d'emmagatzematge

- Els missatges curts són enviats generalment amb `MPI_Bsend`
- Els missatges llargs són enviats generalment amb `MPI_Ssend`

Recepció Estàndard

```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

Implementa el model de recepció amb "rendezvous": el receptor es bloqueja fins que el missatge arriba



- Ineficient: el procés receptor queda bloquejat sense fer res útil

Primitives d'Enviament No Bloquejants

```
MPI_Isend(buf, count, datatype, dest, tag, comm, req)
```

S'inicia l'enviament, però l'emissor no es bloqueja

- Té un argument addicional (`req`)
- Per a reutilitzar el buffer és necessari assegurar-se que l'enviament s'ha completat

Exemple

```
MPI_Isend(A, n, MPI_DOUBLE, dest, tag, comm, &req);
...
/* Comprovar que l'enviament ha acabat,
   amb MPI_Test o MPI_Wait */
A[10] = 2.6;
```

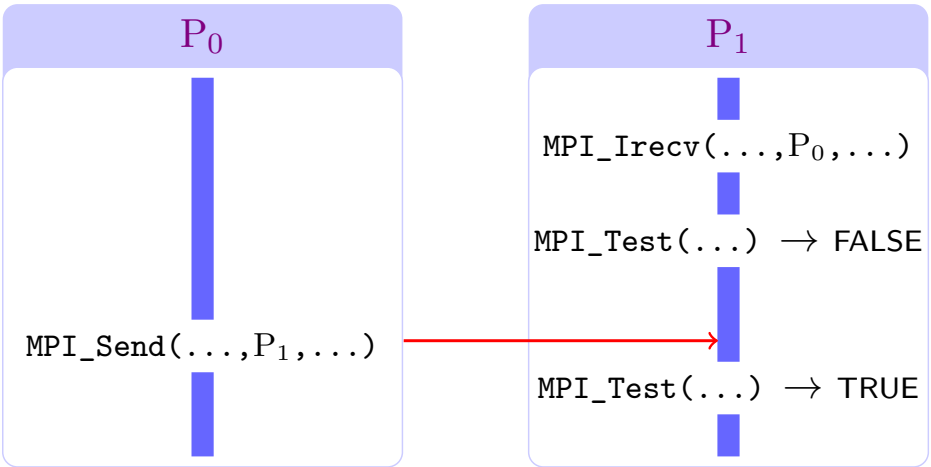
- Solapament de comunicació i càlcul sense còpia extra
- Inconvenient: programació més difícil

Recepció No Bloquejant

```
MPI_Irecv(buf, count, type, src, tag, comm, req)
```

S'inicia la recepció, però el receptor no es bloqueja

- Es substitueix l'argument `stat` per `req`
- És necessari comprovar després si el missatge ha arribat



- Avantatge: solapament de comunicació i càlcul
- Inconvenient: programació més difícil

Operacions Combinades

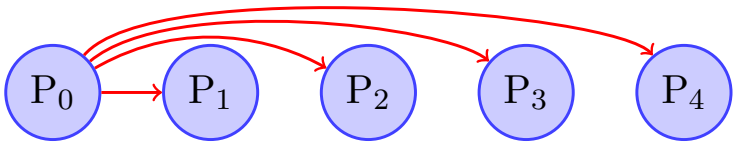
```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
  recvbuf, recvcount, recvtype, source, recvtag, comm,
                          status)
```

Realitza una operació d'enviament i recepció al mateix temps (no necessàriament amb el mateix procés)

```
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag,
              source, recvtag, comm, status)
```

Realitza una operació d'enviament i recepció al mateix temps sobre la mateixa variable

Exemple – Difusió



Difusió d'un valor numèric des de P_0

```
double val;
MPI_Status status;
int p, rank, i;

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  read_value(&val);     /* valor a difondre */
  for (i=1; i<p; i++)
    MPI_Send(&val,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
} else {
  MPI_Recv(&val,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
}
```

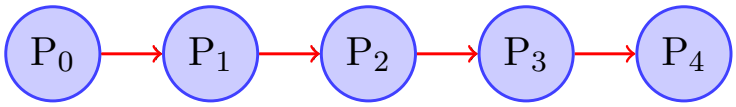
Exemple – Desplaçament en Malla 1-D (1)

Cada procés ha d'enviar la seua dada al veí dret i substituir-ho per la dada que rep del veí esquerre

Desplaçament en Malla 1-D – versió trivial

```
if (rank == 0) {
  MPI_Send(&val, 1, MPI_DOUBLE, rank+1, 0, comm);
} else if (rank == p-1) {
  MPI_Recv(&val, 1, MPI_DOUBLE, rank-1, 0, comm, &status);
} else {
  MPI_Send(&val, 1, MPI_DOUBLE, rank+1, 0, comm);
  MPI_Recv(&val, 1, MPI_DOUBLE, rank-1, 0, comm, &status);
}
```

Inconvenient: Seqüencialització - les comunicacions es realitzen (probablement) seqüencialment, sense concurrència



Exemple – Desplaçament en Malla 1-D (2)

En alguns casos, la programació es pot simplificar utilitzant processos nuls

Desplaçament en Malla 1-D – processos nuls

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;

MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
```

L'enviament al procés `MPI_PROC_NULL` finalitza de seguida; la recepció d'un missatge del procés `MPI_PROC_NULL` no rep res i finalitza de seguida

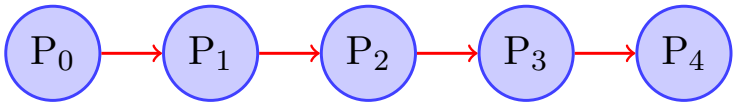
Aquesta versió no resol el problema de la seqüencialització

Exemple – Desplaçament en Malla 1-D (3)

Solució a la seqüencialització: Protocol Parells-Imparells

Desplaçament en Malla 1-D – parells-imparells

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;

if (rank%2 == 0) {
  MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
  MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
} else {
  MPI_Recv(&tmp, 1, MPI_DOUBLE, prev, 0, comm, &status);
  MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
  val = tmp;
}
```

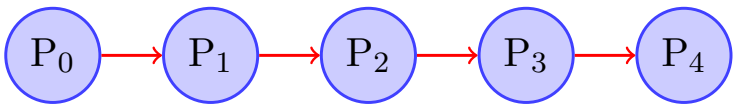


Exemple – Desplaçament en Malla 1-D (4)

Solució a la seqüencialització: Operacions Combinades

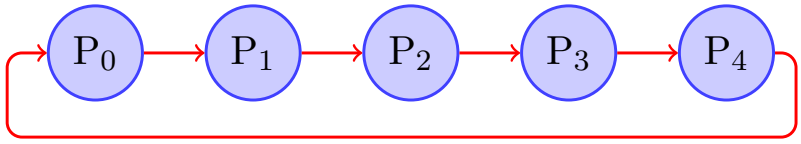
Desplaçament en Malla 1-D – sendrecv

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;

MPI_Sendrecv_replace(&val, 1, MPI_DOUBLE, next, 0, prev, 0,
                     comm, &status);
```



Exemple – Desplaçament en Anell

En el cas de l'anell, tots els processos han d'enviar i rebre



Desplaçament en Anell – versió trivial

```
if (rank == 0) prev = p-1;
else prev = rank-1;
if (rank == p-1) next = 0;
else next = rank+1;

MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
```

Es produirà interbloqueig en el cas d'enviament síncron
Solucions: protocol pares-imparells o operacions combinades

Apartat 3

Comunicació Col·lectiva

- Sincronització
- Difusió
- Repartiment
- Reducció

Operacions de Comunicació Col·lectiva

Involucren a tots els processos d'un grup (comunicador) – tots ells deuen executar l'operació

Operacions disponibles:

- Sincronització (*Barrier*)
- Difusió (*Bcast*)
- Repartiment (*Scatter*)
- Recollida (*Gather*)
- Multi-recollida (*Allgather*)
- Tots a tots (*Alltoall*)
- Reducció (*Reduce*)
- Prefixació (*Scan*)

Aquestes operacions solen tenir com a argument un procés (`root`) que realitza un paper especial

Prefix "`All`": Tots els processos reben el resultat

Sufix "`v`": La quantitat de dades en cada procés és diferent

Sincronització

```
MPI_Barrier(comm)
```

Operació pura de sincronització

- Tots els processos de `comm` es detenen fins que tots ells han invocat aquesta operació

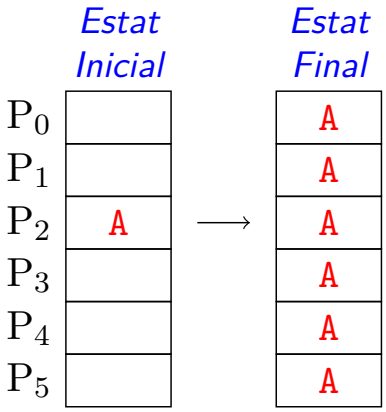
Exemple – mesurament de temps

```
MPI_Barrier(comm);
t1 = MPI_Wtime();
/*
    ...
*/
MPI_Barrier(comm);
t2 = MPI_Wtime();

if (!rank) printf("Temps transcorregut: %f s.\n", t2-t1);
```

Difusió

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

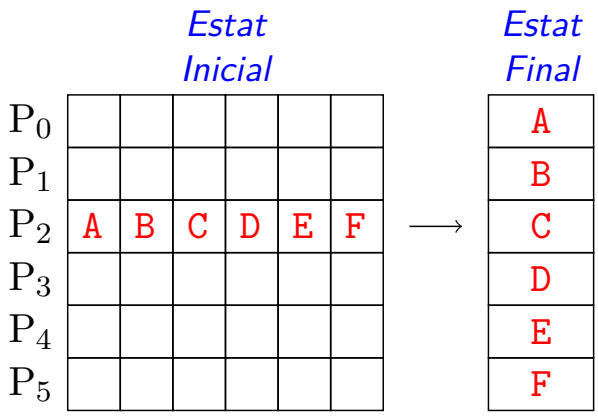
El procés `root` difon a la resta de processos el missatge definit pels 3 primers arguments



Repartiment

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, comm)
```

El procés `root` distribueix una sèrie de fragments consecutius del buffer a la resta de processos (incloent ell mateix)



Versió asimètrica: `MPI_Scatterv`

Repartiment: Exemple

El procés P_0 reparteix un vector de 15 elements (a) entre 3 processos que reben les dades en el vector b

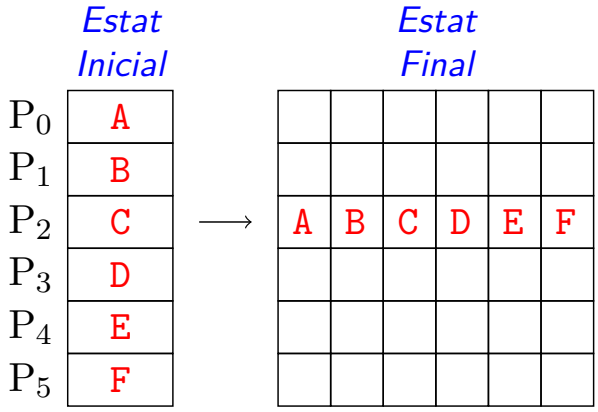
Exemple de repartiment

```
int main(int argc, char *argv[])
{
  int i, myproc;
  int a[15], b[5];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myproc);
  if (myproc==0) for (i=0;i<15;i++) a[i] = i+1;

  MPI_Scatter(a, 5, MPI_INT, b, 5, MPI_INT, 0, MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```


Recollida

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
        recvcount, recvtype, root, comm)
```

És l'operació inversa de `MPI_Scatter`: Cada procés envia un missatge a `root`, el qual ho emmagatzema de forma ordenada d'acord a l'índex del procés en el buffer de recepció

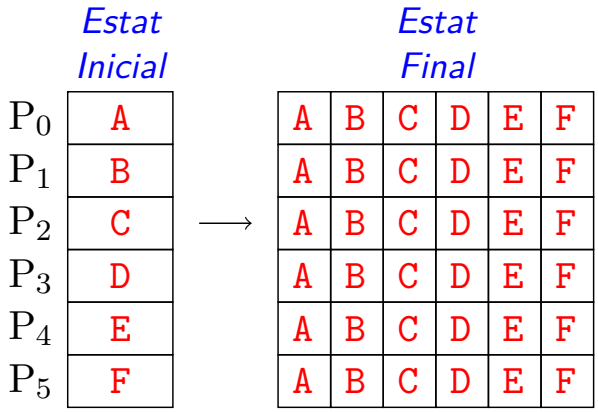


Versió asimètrica: `MPI_Gatherv`

Multi-Recollida

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,
              recvcount, recvtype, comm)
```

Similar a l'operació `MPI_Gather`, però tots els processos obtenen el resultat

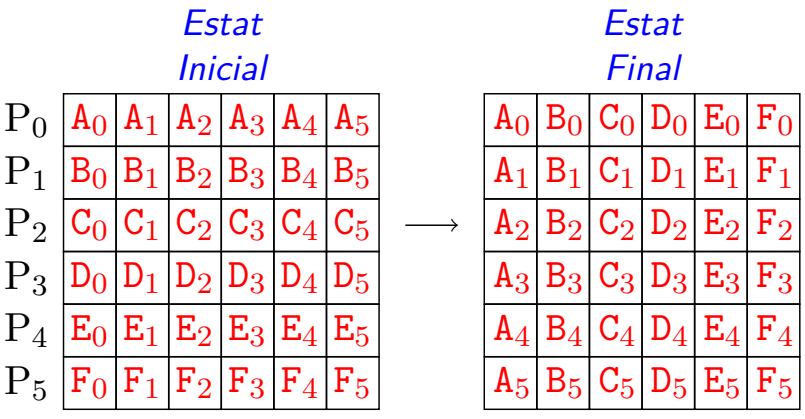


Versió asimètrica: `MPI_Allgatherv`

Tots a Tots

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm)
```

És una extensió de l'operació `MPI_Allgather`, cada procés envia unes dades diferents i rep dades de la resta



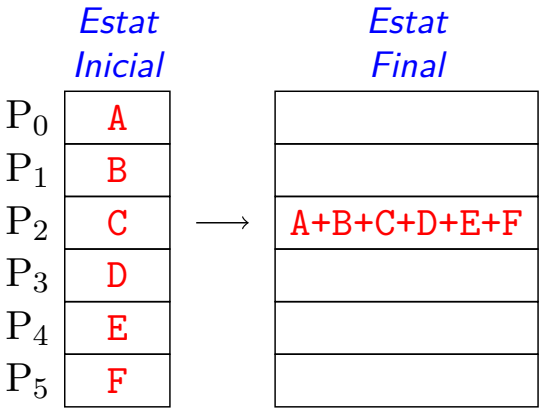
Versió asimètrica: `MPI_Alltoallv`

Reducció

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root,
                                                comm)
```

Similar a `MPI_Gather`, però en lloc de concatenació, es realitza una operació aritmètica o lògica (suma, max, and, ..., o definida per l'usuari)

El resultat final es retorna en el procés `root`



Multi-Reducció

```
MPI_Allreduce(sendbuf, recvbuf, count, type, op, comm)
```

Extensió de `MPI_Reduce` en què tots reben el resultat

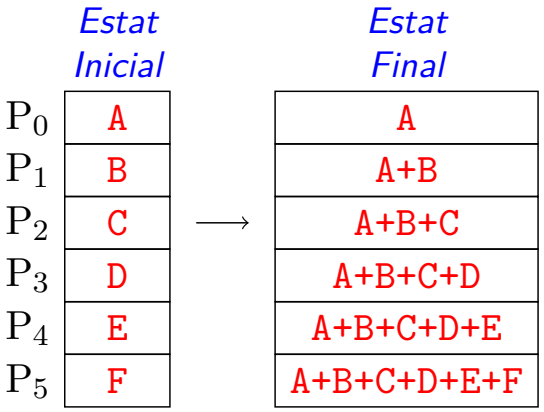
Producte escalar de vectors

```
double par_dot(double local_x[],double local_y[],int local_n)
{
  double local_dot;
  double dot;

  local_dot = seq_dot(local_x, local_y, local_n);
  MPI_Allreduce(&local_dot, &dot, 1, MPI_DOUBLE,
                MPI_SUM, MPI_COMM_WORLD);
  return dot;
}
```

Prefixació

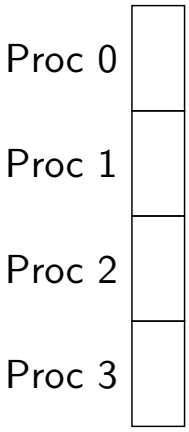
```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)
```

Extensió de les operacions de reducció en què cada procés rep el resultat del processament dels elements dels processos des del 0 fins a ell mateix



Exemple de Prefixació

Donat un vector de longitud N, distribuït entre els processos, on cada procés té n_{local} elements consecutius del vector, es vol obtenir la posició inicial del subvector local



Càlcul de l'índex inicial d'un vector paral·lel

```
int rstart, nlocal, N;

calcula_nlocal(N,&nlocal);    /* per exemple, nlocal=N/p */
MPI_Scan(&nlocal,&rstart,1,MPI_INT,MPI_SUM,comm);
rstart -= nlocal;
```

Apartat 4

Altres Funcionalitats

- Tipus de Dades Derivades

Tipus de Dades Bàsiques

Els tipus de dades bàsiques en llenguatge C són els següents:

`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`

- Per a Fortran existeixen definicions similars
- A més dels anteriors, estan els tipus especials `MPI_BYTE` i `MPI_PACKED`

Dades Múltiples

Es permet l'enviament/recepció de múltiples dades:

- L'emissor indica el nombre de dades a enviar en l'argument `count`
- El missatge ho componen els `count` elements contigus en memòria
- En el receptor, l'argument `count` indica la grandària del buffer – per a saber la grandària del missatge:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype
              datatype, int *count)
```

Aquest sistema no serveix per a:

- Compondre un missatge amb diverses dades de diferent tipus
- Enviar dades del mateix tipus però que no estiguen contigus en memòria

Tipus de Dades Derivades

En MPI es permet definir tipus nous a partir d'altres tipus

El funcionament es basa en les següents fases:

1. El programador defineix el nou tipus, indicant
   - Els tipus dels diferents elements que ho componen
   - El nombre d'elements de cada tipus
   - Els desplaçaments relatius de cada element
2. Es registra com un nou tipus de dades MPI (`commit`)
3. Des de llavors, es pot usar per a crear missatges com si fóra un tipus de dades bàsic
4. Quan no es va a usar més, el tipus es destrueix (`free`)

Avantatges:

- Simplifica la programació quan es repeteix moltes vegades
- No hi ha còpia intermèdia, es compacta només en el moment de l'enviament

Tipus de Dades Derivades Regulars

```
MPI_Type_vector(count, length, stride, type, newtype)
```

Crea un tipus de dades homogeni i regular a partir d'elements d'un *array* equiespaiats

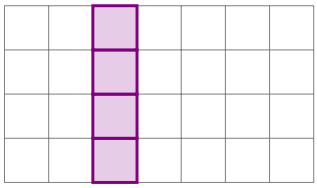
1. De quants blocs es composa (`count`)
2. De quina longitud són els blocs (`length`)
3. Quina separació hi ha entre un element d'un bloc i el mateix element del següent bloc (`stride`)
4. De quin tipus són els elements individuals (`type`)

Constructors relacionats:

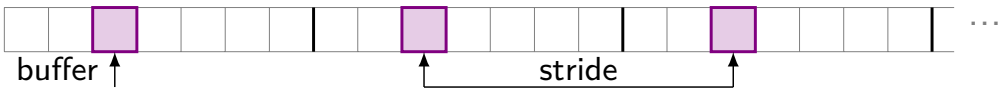
- `MPI_Type_contiguous`: elements contigus
- `MPI_Type_indexed`: longitud i desplaçament variable

Tipus de Dades Derivades Regulars: Example

Volem enviar una *columna* d'una matriu `A[4][7]`



En C, els *arrays* bidimensionals s'emmagatzemen per files




```
double A[4][7];
MPI_Datatype columna;
MPI_Type_vector(4, 1, 7, MPI_DOUBLE, &columna);
MPI_Type_commit(&columna);
if (my_rank == 0) {          /* envia la 3ª columna */
  MPI_Send(&A[0][2], 1, columna, 1, 0, comm);
} else {
  MPI_Recv(&A[0][2], 1, columna, 0, 0, comm, &status);
}
```