Programación en MPI

Índice

Índice

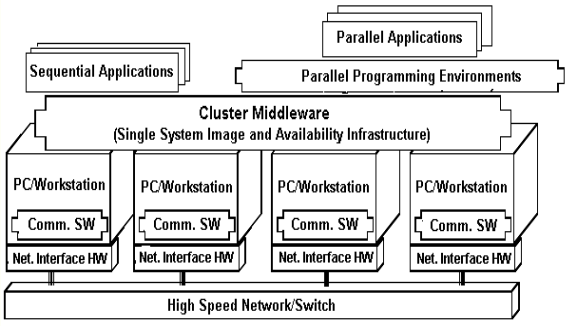
Máquinas de memoria distribuida

- Estas máquinas surgen de forma natural al conectar distintas máquinas en red y ponerlas a cooperar.
- Comunicación y sincronización a través de paso de mensajes (MPI): No comparten memoria
- La *red* es clave para un buen rendimiento

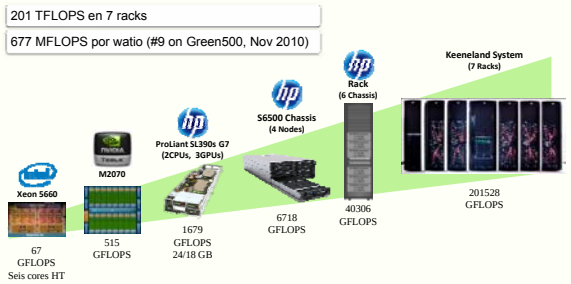
Arquitectura

- El esquema general se basa en varios nodos, multiprocesador (cada uno multicore), conectados por una **red de alto rendimiento**.
- *Single System Image* (SSI) para proporcionar una imagen unificada y simplificada del cluster
- No comparten el espacio de direcciones (excepto los NUMA).

Arquitectura de un cluster



Ejemplo de cluster HPC



Redes

- Dos factores críticos: **Ancho de banda y latencia**
- Ancho de banda: 10-100 Gb/s. Latencia ≈1 µs
- Infiniband: Enlaces 1X, 4X o 12X, Data rate (Gb/s 1X): SDR (2), DDR (4), QDR (8), FDR (10), FDR (14), EDR (25)
- Gigabit: 10GbE, 40GbE, 100GbE
- Myrinet, en desuso: 10 Gb/s
- Cerca de las prestaciones de buses modernos:
  - PCI Express 3.0 (x32): 256 Gbit/s
  - QuickPath Interconnect (4.0 GHz): 256 Gbit/s
  - HyperTransport 3.1 (3.2 GHz, 32-pair): 409.6 Gbit/s

MPI

- Es un estándar para paso de mensajes.
- Las aplicaciones son fácilmente portables.
- Es útil para MPP, NOW, Beowulf, NUMA, ...
- No es un nuevo lenguaje, sino una biblioteca que se añade a lenguajes ya existentes (C, C++, F90).

MPI

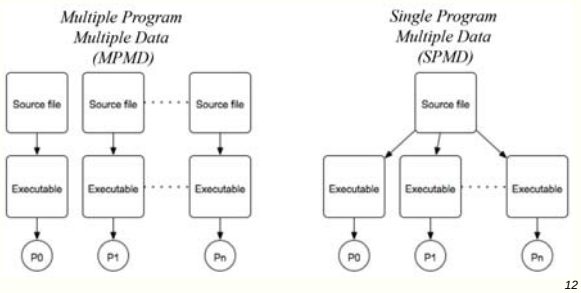
- Hace más hincapié en el rendimiento, y no en la heterogeneidad o la tolerancia a fallos (MPP).
- Han añadido aspectos de E/S: MPI-IO.
- Permite heterogeneidad.
- En MPI-2:
  - Estandariza el arranque de las aplicaciones. Quitaba portabilidad.
  - Añade aspectos dinámicos, permite arrancar procesos durante la ejecución. Poco usado, lo normal es pedir unos recursos durante un tiempo, sin cambios

MPI

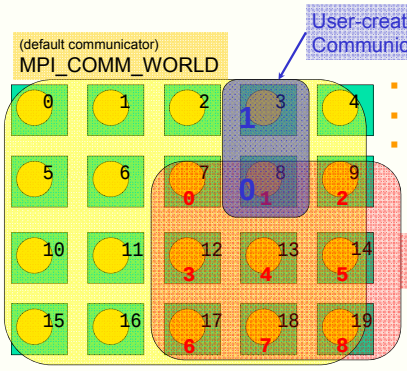
- Se puede combinar con *threads*, **MPI multithread**, o con otras bibliotecas, si están preparadas.
- Atención al soporte dado si se usa **MPI multithread**
  - MPI THREAD SINGLE: Sólo un thread
  - MPI THREAD FUNNELED: Multithread, pero sólo uno (el maestro) puede llamar a la biblioteca MPI
  - MPI THREAD SERIALIZED: Multithread, pero no admite llamadas simultáneas, se serializan
  - MPI THREAD MULTIPLE: Multithread, sin restricciones.
- MPI_Init_thread(required, provided)**:** Informa del nivel de soporte
- Se puede y suele combinar con **OpenMP**
- Ejemplo: Servicio distribuido siempre a la escucha

Esquema SPMD

- Al compartir el código fuente suele ser más legible y fácil de mantener.



Comunicadores de MPI



- Size
- Rank
- Útil para varias apps colaborando

Credit: Allan Snavely

Funciones básicas

¡¡¡Lo mínimo para hacer una aplicación MPI son seis rutinas!!!

- Arranque y parada.
- Envío y recepción bloqueantes.
- Envío y recepción No bloqueantes
- Sincronización (barreras)
- Envío y recepción múltiple

Arranque y parada

```
int MPI_Init(int *argc, char **argv);
    IN    argc
    IN    argv
int MPI_Finalize(void);
int MPI_Comm_size(MPI_Comm comm, int *size);
    IN    comm          communicator
    OUT   size          number of tasks in "comm"
int MPI_Comm_rank(MPI_Comm comm,int *rank);
    IN    comm          communicator
    OUT   rank          rank of the calling task
```

- Misma variable, al menos de nombre (rank) pero distinta memoria y por tanto distinto valor

Envío y recepción bloqueantes

```
int MPI_Send(void* buf, int count, MPI_Datatype
    datatype,int dest, int tag, MPI_Comm comm) ;
    IN    buf       initial address of send buffer
    IN    count     number of entries to send
    IN    datatype  datatype of each entry
    IN    dest      rank of destination
    IN    tag       message tag
    IN    comm      communicator
```


Envío y recepción bloqueantes

```
int MPI_Recv(void* buf, int count, MPI_Datatype
    datatype, int source, int tag, MPI_Comm comm,
    MPI_Status *status);
    OUT   buf      initial address of receive buff
    IN    count    max # of entries to receive
    IN    datatype datatype of each entry
    IN    source   rank of source
    IN    tag      message tag
    IN    comm     communicator
    OUT   status   return status
```

- Los mensajes se pueden filtrar (elegir) por origen, etiqueta y/o comunicador. El uso de comodines permite recibir el "siguiente" mensaje: MPI_ANY_TAG y MPI_ANY_SOURCE
- Los *tags* permiten identificar los mensajes, establece "tipos de mensaje" (datos, control, errores,…)
- No se puede suponer ningún orden de llegada!!!.
- Status indica estado, *source* y *tag*

Ejemplo: Hello World!

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv) {
  int rank;
  char msg[20];
  MPI_Status status;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

...
```

Ejemplo: Hello World!

```
if (rank==0) {
  printf ("I am master. Sending the message.\n\n");
  strcpy(msg,"Hello World!");
  MPI_Send(msg, 13, MPI_CHAR, 1, 100, MPI_COMM_WORLD);
} else {
  printf ("I am slave. Receiving the message.\n");
  MPI_Recv(msg, 13, MPI_CHAR, 0, 100, MPI_COMM_WORLD,
   &status);
  printf ("The message is: %s\n", msg);
}

MPI_Finalize();
}
```

19

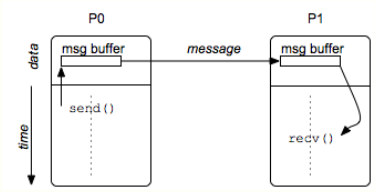
MPI Comunicación punto a punto

- Dos modos de comunicación:
  - Síncrona o bloqueante: En teoría no retorna hasta que se completa la operación (no siempre es así). Se pueden modificar los datos de origen sin problema.
  - Asíncrona o no bloqueante : Retorna enseguida, indica que se desea iniciar esa operación. No se puede modificar los datos de origen
- MPI proporciona cuatro versiones para cada modo.
  - Standard: Send
  - Synchronous: Ssend
  - Buffered: Bsend
  - Ready: Rsend

20

Synchronous & Buffered Send

- Buffered (MPI_Bsend). Copia internamente el mensaje en la biblioteca y retorna al llamante
  - No hay comunicación real hasta que se hace un *receive*.
  - V: Desacopla envío y recepción, el emisor no espera
  - I: Añade el coste de copiar los datos y la gestión del buffer depende del programador.



21

Synchronous & Ready & Standard Send

- Synchronous (MPI_Ssend). El envío espera a que alguien pida recibir los datos (bloqueante)
  - Entonces pone los datos en la red y espera confirmación
  - No hay buffering adicional.
  - V: Seguro y portable. I: Larga espera
- Ready (MPI_Rsend). Válido sólo si el receptor está listo
  - V: Más eficiente ya que no hay dialogo. I: Peligroso
- Standard (MPI_Send). La biblioteca dedice si espera (bloqueante) o copia en memoria el mensaje (localmente bloqueante) y retorna. Cambia con la implementación
  - Según el tamaño del mensaje, para pequeños usa buffer, con grandes espera.
  - V: Genérico para la mayoría de casos.

22

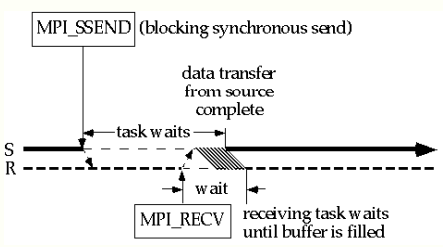
Modos de comunicación punto a punto

Modo comunicación	Bloqueante	No bloqueante
Synchronous Send	***MPI_Ssend***	***MPI_Issend***
Ready Send	***MPI_Rsend***	***MPI_Irsend***
Buffered Send	***MPI_Bsend***	***MPI_Ibsend***
Standard Send	***MPI_Send***	***MPI_Isend***
Recepción (recv)	**MPI_Recv**	**MPI_Irecv**

23

MPI Synchronous Send: MPI_Ssend

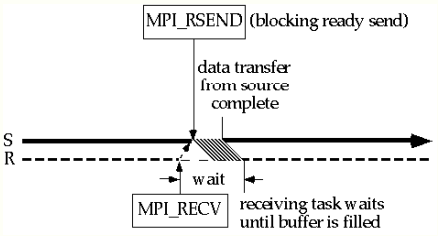
- Espera al receptor



24

MPI Blocking Ready Send: MPI_Rsend

- El receptor ya está listo para recibir (sino *error*)

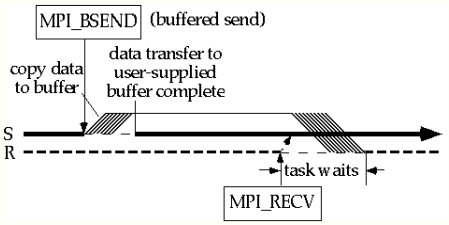


25

MPI Buffered Send: MPI_Bsend

- El programador gestiona el buffer. Se copia el mensaje y retorna

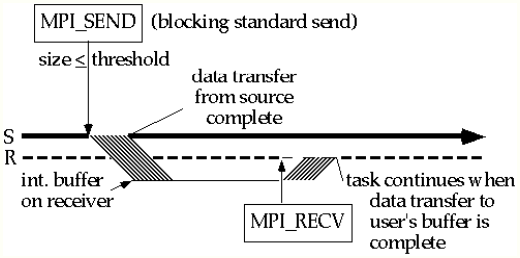
```
MPI_Buffer_attach(void *buf, int size)
```



26

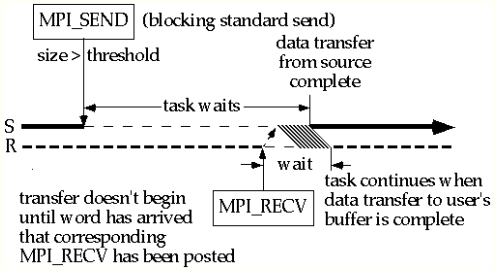
MPI Blocking Standard Send Small Message Size

- Límite, *threshold*, depende de implementación



27

MPI Blocking Standard Send Large Message Size



28

Envío y recepción No bloqueantes

```
int MPI_Isend(void* buf, int count, MPI_Datatype
    datatype, int dest, int tag, MPI_Comm comm,
    MPI_Request *request);
    IN    buf       initial address of send buffer
    IN    count     number of entries to send
    IN    datatype  datatype of each entry
    IN    dest      rank of destination
    IN    tag       message tag
    IN    comm      communicator
    OUT   request   request handle
```

29

Envío y recepción No bloqueantes

```
int MPI_Irecv(void* buf, int count, MPI_Datatype
    datatype, int source, int tag, MPI_Comm comm,
    MPI_Request*request)
    OUT   buf       initial address of receive buff
    IN    count     max # of entries to receive
    IN    datatype  datatype of each entry
    IN    source    rank of source
    IN    tag       message tag
    IN    comm      communicator
    OUT   request   request handle
```

30

Envío y recepción No bloqueantes

```
int MPI_Wait(MPI_Request *request, MPI_Status *
    status);
    INOUT request      request handle
    OUT   status       status object
int MPI_Test(MPI_Request *request, int *flag,
      MPI_Status *status);
    INOUT request      request handle
    OUT   flag         true if operation completed
    OUT   status       status object
int MPI_Request_free(MPI_Request *request) ;
    INOUT request      request handle
```


Envío y recepción No bloqueantes

- La idea es solapar cómputo y comunicación.

  Ej: recibir la siguiente tarea a hacer mientras se procesa la actual.

```
Image * Curr_img, *Next_img;
MPI_Recv(Curr_img,…);
while (status.tag != END) {
    MPI_Irecv(Next_img,…,Next_req); // No espera!!!
    Compute_img(Curr_img); // Time consuming!!!
    MPI_Send(results,…);   // Copy data and return
    MPI_Wait(Next_req);    // Return quickly
    Curr_img = Next_req;   // Copying pointers
}
```


Alternativas al wait

- Espera por todas las operaciones pendientes

  ```
  int MPI_Waitall(int count, MPI_Request
     *array_of_requests, MPI_Status
     *array_of_statuses)
  ```

- Espera por la primera que termine (indica con index)

  ```
  int MPI_Waitany(int count, MPI_Request
     *array_of_requests, int *index, MPI_Status
     *status)
  ```

- Espera por al menos una operación. El número de operaciones que terminó se indica con outcount y sus indices y status son devueltos.

  ```
  int MPI_Waitsome(int incount, MPI_Request
     *array_of_requests,  int *outcount,
     int *array_of_indices, MPI_Status
     *array_of_statuses)
  ```


Alternativas al Test

- Comprueba si todas las operaciones terminaron (flag=1)

  ```
  int MPI_Testall(int count, MPI_Request
     *array_of_requests, int *flag,MPI_Status
     *array_of_statuses)
  ```

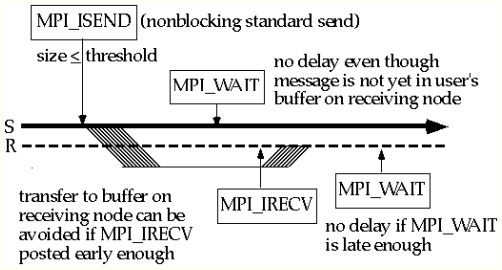
- Comprueba si alguna terminó (flag=1), e indica su índice

  ```
  int MPI_Testany(int count, MPI_Request
     *array_of_requests, int *index, int *flag,
     MPI_Status *status)
  ```

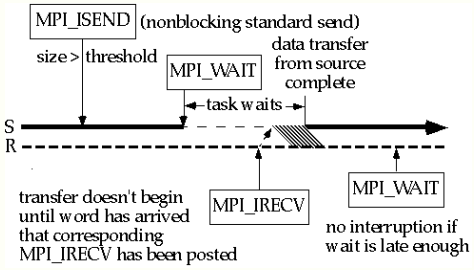
- Comprueba si algunas terminaron (flag=1), e indica cuántas (outcount) y sus índices

  ```
  int MPI_Testsome(int incount, MPI_Request
     *array_of_requests, int *outcount, int*
     array_of_indices, MPI_Status
     *array_of_statuses)
  ```


MPI Nonblocking Standard Send Small Message Size



MPI Nonblocking Standard Send Large Message Size



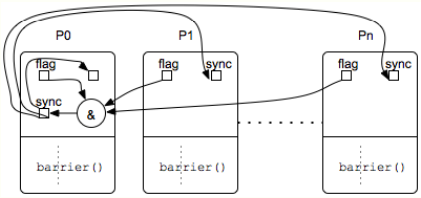
Comunicaciones collectivas

- Comunicación entre un grupo de procesos, identificado por el comunicador. No usa *tags*.
- Tres clases:
  - Sincronización: Barreras
  - Transferencia de datos:
    - Broadcast
    - Scatter
    - Gather
    - All-to-all
  - Cómputo global
    - Reduce
    - Scan

Barrera de sincronización

- Barrera

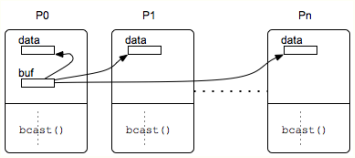
```
int MPI_Barrier(MPI_Comm comm)
      IN    comm     communicator
```



Envío y recepción múltiples

- Broadcast: Envío de datos a todos los nodos desde el *root*.

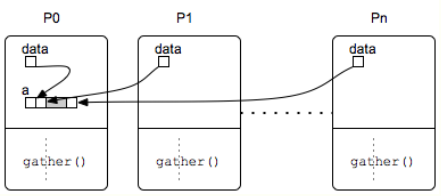
```
int MPI_Bcast(void* buffer, int count, MPI_Datatype
    datatype, int root, MPI_Comm comm )
    INOUT buffer   starting address of buffer
    IN    count    number of entries in buffer
    IN    datatype data type of buffer
    IN    root     rank of broadcast root
    IN    comm     communicator
```



Envío y recepción múltiples

- Recolección de resultados: Todos los nodos mandan resultado al root.

```
MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
    sendtype, void* recvbuf, int recvcount, MPI_Datatype
    recvtype, int root, MPI_Comm comm)
```



Envío y recepción múltiples

- Recolección de resultados: Todos los nodos mandan resultado al root.

```
MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
    sendtype, void* recvbuf, int recvcount, MPI_Datatype
    recvtype, int root, MPI_Comm comm)
    IN    sendbuf   starting address of send buffer
    IN    sendcount number of elements in send buffer
    IN    sendtype  data type of send buffer elements
    OUT   recvbuf   address of receive buffer
    IN    recvcount number of elements for any receive
    IN    recvtype  data type of recv buffer elements
    IN    root      rank of reciving process
    IN    comm      communicator
```


Envío y recepción múltiples

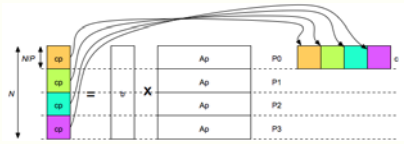
- Recolección de resultados: Todos los nodos mandan resultado al root. Cada uno manda un número distinto de elementos.

```
MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype
    sendtype, void* recvbuf, int *recvcounts, int
    *displs, MPI_Datatype recvtype, int root, MPI_Comm
    comm)
    IN    sendbuf    starting address of send buffer
    IN    sendcount  number of elements in send buffer
    IN    sendtype   datatype of send buffer elements
    OUT   recvbuf    address of receive buffer
    IN    recvcounts integer array
    IN    displs     integer array of displacements
    IN    recvtype   data type of recv buffer elements
    IN    root       rank of receiving process
    IN    comm       communicator
```

MPI Gather: Ejemplo

- Producto vect-Mat Axb=c

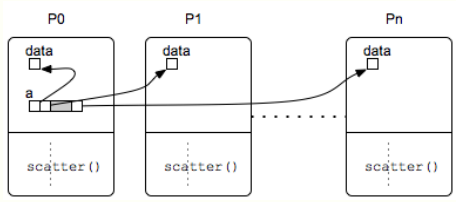
```
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];
for (i = 1; i < N/P; i++) {
    cp[i] = 0;
    for (k = 0; k < N; k++)
        cp[i] = cp[i] + Ap[i][k] * b[k];
}
MPI_Gather(cp, N/P, MPI_Float, c, N/P, MPI_Float,
    root, MPI_COMM_WORLD); // root = 0
```



Envío y recepción múltiples

- Envío múltiple: Cada nodo recibe **parte** de los datos

```
MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
    sendtype, void* recvbuf, int recvcount, MPI_Datatype
    recvtype,int root, MPI_Comm comm)
```



Envío y recepción múltiples

- Envío múltiple: Cada nodo recibe **parte** de los datos

```
MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
    sendtype, void* recvbuf, int recvcount, MPI_Datatype
    recvtype,int root, MPI_Comm comm)
    IN    sendbuf    address of send buffer
    IN    sendcount  nº of elements send to each process
    IN    sendtype   datatype of send buffer elements
    OUT   recvbuf    address of receive buffer
    IN    recvcount  number of elements in receive buffer
    IN    recvtype   data type of recv buffer elements
    IN    root       rank of sending process
    IN    comm       communicator
```


Envío y recepción múltiples

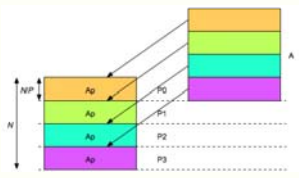
- Envío múltiple: Cada nodo recibe **parte** de los datos. Número de elementos a recibir distinto en cada nodo

```
MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
    MPI_Datatype sendtype, void* recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
    IN    sendbuf    address of send buffer
    IN    sendcounts integer array
    IN    displs     integer array of displacements
    IN    sendtype   datatype of send buffer elements
    OUT   recvbuf    address of receive buffer
    IN    recvcount  number of elements in receive buffer
    IN    recvtype   data type of recv buffer elements
    IN    root       rank of sending process
    IN    comm       communicator
```


MPI Scatter: Ejemplo

- Distribuir una matriz NxN entre P procesos

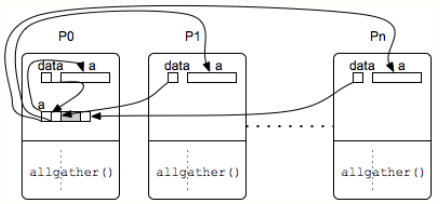
```
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];
MPI_Scatter(A, N/P*N, MPI_Float, Ap, N/P*N,
    MPI_Float, 0 ,MPI_COMM_WORLD); // root = 0;
```



Envío y recepción múltiples

- Recolección de resultados: **Todos** los nodos reciben **todos** los datos. Es un *gather* más un *broadcast* del resultado

```
MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype
    sendtype, void* recvbuf, int recvcount, MPI_Datatype
    recvtype, MPI_Comm comm)
```



Envío y recepción múltiples

- Recolección de resultados: **Todos** los nodos reciben **todos** los datos. Es un *gather* más un *broadcast* del resultado

```
MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype
    sendtype, void* recvbuf, int recvcount, MPI_Datatype
    recvtype, MPI_Comm comm)
    IN    sendbuf   starting address of send buffer
    IN    sendcount number of elements in send buffer
    IN    sendtype  datatype of send buffer elements
    OUT   recvbuf   address of receive buffer
    IN    recvcount number of elements received from any
    process
    IN    recvtype  data type of recv buffer elements
    IN    comm      communicator
```

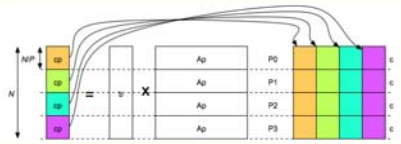

Envío y recepción múltiples

- Recolección de resultados: **Todos** los nodos reciben **todos** los datos. Cada nodo envía un número distinto de elementos

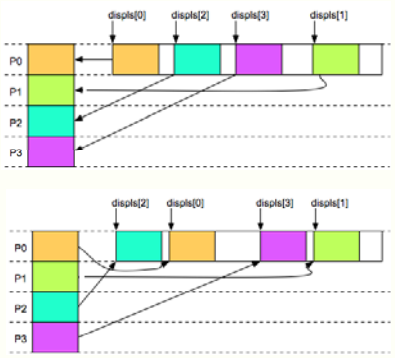
```
MPI_Allgatherv(void* sendbuf, int sendcount,
    MPI_Datatype sendtype, void* recvbuf, int
    *recvcounts, int *displs, MPI_Datatype recvtype,
    MPI_Comm comm)
    IN    sendbuf    starting address of send buffer
    IN    sendcount  number of elements in send buffer
    IN    sendtype   datatype of send buffer elements
    OUT   recvbuf    address of receive buffer
    IN    recvcounts integer array
    IN    displs     integer array of displacements
    IN    recvtype   data type of recv buffer elements
    IN    comm       communicator
```


MPI AllGather: Ejemplo

```
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];
for (i = 1; i < N/P; i++) {
    cp[i] = 0;
    for (k = 0; k < N; k++)
        cp[i] = cp[i] + Ap[i][k] * b[k];
}
MPI_AllGather(cp, N/P, MPI_Float, c, N/P, MPI_Float,
MPI_COMM_WORLD);
```



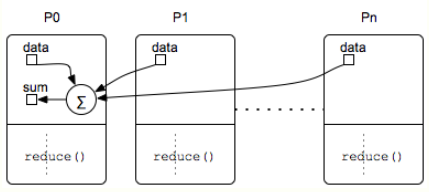
MPI Scatterv and (All)Gatherv



Envío y recepción múltiples

- Recolección de resultados: Todos los nodos mandan resultados al root que los combina (reduce) a un único valor

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
    comm)
```



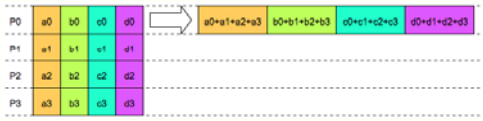
Envío y recepción múltiples

- Recolección de resultados: Todos los nodos mandan resultados al root que los combina (reduce) a un único valor

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
    comm)
    IN    sendbuf  address of send buffer
    OUT   recvbuf  address of receive buffer (at root)
    IN    count    number of elements in send buffer
    IN    datatype data type of elements of send buffer
    IN    op       reduce operation
    IN    root     rank of root process
    IN    comm     communicator
```

MPI Reduce: Ejemplo

```
float abcd[4], sum[4];
MPI_Reduce(abcd, sum, 4, MPI_Float, root, MPI_SUM,
    MPI_COMM_WORLD); // root = 0
```

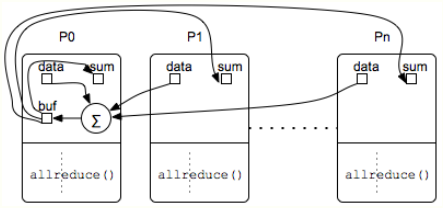


abcd en cada nodo sum en root

Envío y recepción múltiples

- Recolección de resultados: **Todos** los nodos reciben **todos** los datos. Es un reduce más un broadcast del resultado

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int
    count, MPI_Datatype datatype, MPI_Op op, MPI_Comm
    comm)
```



Envío y recepción múltiples

- Recolección de resultados: **Todos** los nodos reciben **todos** los datos. Es un reduce más un broadcast del resultado

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int
    count, MPI_Datatype datatype, MPI_Op op, MPI_Comm
    comm)
    IN    sendbuf  address of send buffer
    OUT   recvbuf  address of receive buffer (at root)
    IN    count    number of elements in send buffer
    IN    datatype data type of elements of send buffer
    IN    op       reduce operation
    IN    comm     communicator
```


MPI AllReduce: Ejemplo

```
float abcd[4], sum[4];
MPI_AllReduce(abcd, sum, 4, MPI_Float, MPI_SUM,
    MPI_COMM_WORLD);
```

Envío y recepción múltiples

- Recolección de resultados: Todos los nodos mandan resultados a todos los procesos que los combina (reduce) a un único valor. Proceso i recibe y combina datos del proceso 0 al i.

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
    IN    sendbuf  address of send buffer
    OUT   recvbuf  address of receive buffer (at root)
    IN    count    number of elements in send buffer
    IN    datatype data type of elements of send buffer
    IN    op       reduce operation
    IN    comm     communicator
```


Operandos para Reduce y Scan

MPI_OP	Operation	C	Fortran
MPI_MAX	maximum	integer, float	integer, real, complex
MPI_MIN	minimum	integer, float	integer, real, complex
MPI_SUM	sum	integer, float	integer, real, complex
MPI_PROD	product	integer, float	integer, real, complex
MPI_LAND	logical and	integer	logical
MPI_BAND	bit-wise and	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	logical or	integer	logical
MPI_BOR	bit-wise or	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	logical xor	integer	logical
MPI_BXOR	bit-wise xor	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	max val and loc	float, double, long double	real, complex,double precision
MPI_MINLOC	min val and loc	float, double, long double	real, complex,double precision

Puede crear sus Operadores con MPI_Op_create

Ejemplos

- Veamos a continuación algunos ejemplos:
  - El primer nodo envía un mensaje al siguiente y espera que le vuelva. Cada nodo lo reenvía a su vecino.
  - Todos los nodos envían un mensaje al siguiente y esperan que le vuelva. Reenvían los mensajes que le llegan a su vecino.
  - Uso de primitivas síncronas y asíncronas
  - Cálculo del número PI

Funciones avanzadas

- Empaquetar/desempaquetar
- Definir nuevos tipos.
- Comunicación persistente
- Mensajes de tamaño desconocido
- Creación de nuevos comunicadores
- Creación dinámica de procesos

Empaquetado/desempaquetado

- El coste de enviar varios mensajes pequeños es muy alto, por la latencia, mejor agruparlo y mandar uno:

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype
    datatype, void *outbuf, int outsize, int
    *position, MPI_Comm comm)

    IN    inbuf    input buffer start
    IN    incount  number of input data items
    IN    datatype datatype of each input data item
    OUT   outbuf   output buffer start
    IN    outsize  output buffer size, in bytes
    INOUT position current position in buffer(bytes)
    IN    comm     communicator for packed message
```


Empaquetado/desempaquetado

```
MPI_Unpack(void* inbuf, int insize, int * position
    void *outbuf, int outcount, MPI_Datatype *
    datatype, MPI_Comm comm)

    IN    inbuf    input buffer start
    IN    insize   size of input buffer (bytes)
    INOUT position current position bytes)
    OUT   outbuf   output buffer start
    IN    outcount number of items to be unpacked
    IN    datatype datatype of each output data item
    IN    comm     communicator for packed message
```


Empaquetado/desempaquetado

- Para asegurar la portabilidad no suponer nada de los tamaños y preguntar en cada caso cual es:

```
int MPI_Pack_size(int incount, MPI_Datatype datatype,
    MPI_Comm comm, int *size)

    IN    incount  count argument to packing call
    IN    datatype datatype argument to packing call
    IN    comm     communicator argument to packing call
    OUT   size     upper bound on size of packed
                   message (bytes)
```


Ejemplo pack

```
int count,counts[64], size1, size2, size,position;
char chr[100], *packbuf;
...
// Enviar N caracteres (chr), N no conocido (count)
// Dos mensajes tarda el doble
// allocate local pack buffer

MPI_Pack_size(1, MPI_INT, comm, &size1);
MPI_Pack_size(count, MPI_CHAR, comm, &size2);
size = size1+size2;
packbuf = (char *)malloc(size);

// pack count, followed by count characters

position = 0;
MPI_Pack(&count, 1, MPI_INT, packbuf, size, &position, comm);

// Gracias a position se encadenan uno tras otro

MPI_Pack(chr, count, MPI_CHAR, packbuf, size, &position, comm);
```

Tipos predefinidos

- **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***
- **MPI_BYTE**
- **MPI_PACKED**
- Usar tipos definidos por el usuario ayuda a mejorar la legibilidad ya que el programa se expresa en términos de la aplicación

67

Creación de tipos: contiguo

```
MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)

    IN    count    replication count
    IN    oldtype  old datatype
    OUT   newtype  new datatype
```

68

Creación de tipos: vector

```
MPI_Type_vector(int count, int blocklength, int stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)

    IN   count        number of blocks
    IN   blocklength  nº of elements in each block
    IN   stride       spacing between start of each block,
                      measured as number of elements
    IN   oldtype      old datatype
    OUT  newtype      new datatype

MPI_Type_hvector(int count, int blocklength, int
    stride, MPI_Datatype oldtype, MPI_Datatype
    *newtype)
                        stride is given in bytes
```

69

Creación de tipos: index

```
MPI_Type_indexed(int count, int *array_of_blocklengths,
    int *array_of_displacements, MPI_Datatype oldtype,
    MPI_Datatype *newtype)

    IN    count                 number of blocks
    IN    array_of_blocklengths nº of elemnts per block
    IN    array_of_displacements displacement for each
                                 block, measured as
                                 number of elements
    IN    oldtype               old datatype
    OUT   newtype               new datatype

MPI_Type_hindexed(int count, int *array_of_blocklengths,
    int *array_of_displacements, MPI_Datatype oldtype,
    MPI_Datatype *newtype)

                        displacements is given in bytes
```

70

Creación de tipos: struct

```
MPI_Type_struct(int count, int *array_of_blocklengths,
    MPI_Aint *array_of_displacements, MPI_Datatype
    *array_of_types, MPI_Datatype *newtype)

    IN    count                  number of blocks
    IN    array_of_blocklengths  nº of elemnts per block
    IN    array_of_displacements byte displacement
                                 for each block
    IN    oldtype        type of elements in each block
    OUT   newtype        new datatype
```

71

Comunicación persistente

- Reutiliza la estructura de los mensajes. Siempre asíncrono.
- Habitual que se repita el mismo esquema de comunicación
- No inicia comunicación alguna, construye el mensaje

```
int MPI_Send_init(void* buf, int count,
    MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request);

    IN    buf       initial address of send buffer
    IN    count     number of entries to send
    IN    datatype  datatype of each entry
    IN    dest      rank of destination
    IN    tag       message tag
    IN    comm      communicator
    OUT   request   communication request handle
```

72

Comunicación persistente

```
int MPI_Recv_init(void* buf, int count,
    MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Request *request);
```

OUT	buf	initial address of receive buff
IN	count	max # of entries to receive
IN	datatype	datatype of each entry
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	request	communiction request

Comunicación persistente

```
int MPI_Start(MPI_Request *request)
    INOUT   request  communication request handle

int MPI_Startall(int count, MPI_Request
    *array_of_requests)
    IN count list length
    INOUT array_of_requests array of requests

int MPI_Request_free(MPI_Request *request)
    INOUT request communication request

int MPI_Wait(MPI_Request *request, MPI_Status *status)
    INOUT request  communication request
    OUT   status   status object
```

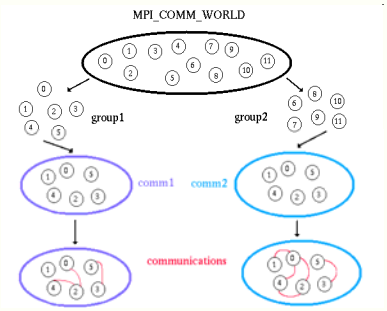

Mensajes de tamaño desconocido

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status
    *status)
    IN    source    rank of source or MPI_ANY_SOURCE
    IN    tag       message tag or MPI_ANY_TAG
    IN    comm      communicator
    OUT   status    status object

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
    MPI_Status *status)
    IN    source    rank of source or MPI_ANY_SOURCE
    IN    tag       message tag or MPI_ANY_TAG
    IN    comm      communicator
    OUT   flag      true if message pending
    OUT   status    status object

int MPI_Get_count(MPI_Status *status,MPI_Datatype datatype, int
    *count);
    IN    status    status object
    IN    datatype  datatype of each entry
    OUT   count     number of entries
```


Comunicadores y grupos



Creación de grupos

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

int MPI_Group_incl(MPI_Group group, int n, int *ranks,
    MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int *ranks,
    MPI_Group *newgroup)
int MPI_Group_range_incl(MPI_Group group, int n, int
    ranges[][3], MPI_Group *newgroup)
int MPI_Group_range_excl(MPI_Group group, int n, int
    ranges[][3], MPI_Group *newgroup)

int MPI_Group_union(MPI_Group group1, MPI_Group
    group2, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group
    group2, MPI_Group *newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group
    group2, MPI_Group *newgroup)
```


Creación de grupos

- pre-defined group:

```
MPI_GROUP_EMPTY: a group with no members.
MPI_GROUP_NULL: constant for invalid group.

int MPI_Group_free(MPI_Group *group)

int MPI_Group_size(MPI_Group group, int *size)

int MPI_Group_compare(MPI_Group group1,MPI_Group
    group2, int *result)

int MPI_Group_rank(MPI_Group group, int *rank)

int MPI_Group_translate_ranks (MPI_Group group1, int
    n, int *ranks1, MPI_Group group2, int *ranks2)
```

Creación de comunicadores

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

int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2,
    int *result)
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
    MPI_Comm *newcomm)
int MPI_Comm_split(MPI_Comm comm, int color, int key,
    MPI_Comm *newcomm)
int MPI_Comm_free(MPI_Comm *comm)
```


Creación de procesos

```
int MPI_Comm_spawn(char *command, char *argv[], int
    maxprocs, MPI_Info info, int root, MPI_Comm comm,
    MPI_Comm *intercomm, int array_of_errcodes[])

    IN command       name of program to be spawned (root)
    IN argv          arguments to command (root)
    IN maxprocs      max. nº of processes to start (root)
    IN info          a set of key-value telling where and
                     how to start the processes (root)
    IN root
    IN comm          intracommunicator
    OUT intercomm    intercommunicator between original
                     group and the newly spawned group
    OUT array_of_errcodes      one code per process
```


Creación de procesos

```
int MPI_Comm_spawn_multiple(int count, char
    *array_of_commands[], char **array_of_argv[], int
    array_of_maxprocs[], MPI_Info array_of_info[], int
    root, MPI_Comm comm, MPI_Comm *intercomm, int
    array_of_errcodes[])

    IN    count             number of commands (root)
    IN    array_of_commands programs to be executed (root)
    IN    array_of_argv     arguments for commands (array of
                            array of strings, root)
    IN    array_of_maxprocs maximum number of processes to
                            start for each command (root)
    IN    array_of_info     info objects telling where and how
                            to start processes (root)
    IN    root
    IN    comm              intracommunicator
    OUT   intercomm         intercommunicator between original
                            group and newly spawned group

    OUT   array_of_errcodes one error code per process
```