MPI
Un estándar de paso de mensajes para Clusters y Workstations

Communications of the ACM, July 1996
J.J. Dongarra, S.W. Otto, M. Snir, and D.W. Walker
Traducido por José Luis Elvira

Message Passing Interface (MPI)

- Librería de paso de mensajes
- Puede agregarse a lenguajes secuenciales (C, Fortran)
- Diseñado por un consorcio integrado por industria, academia, gobierno
- La meta es tener un estándard para el paso de mensajes

Modelo de Programación MPI

- Multiple Program Multiple Data (MPMD)
  - Los procesadores pueden ejecutar diferentes programas (a diferencia de SPMD)
  - Número de procesadores fijo (uno por procesador)
  - No soporta multi-hilos

- Comunicación Punto-a-Punto y colectiva

Funciones básicas de MPI

MPI_INIT	Inicializa MPI
MPI_FINALIZE	Termina la computación
MPI_COMM SIZE	Número of procesos
MPI_COMM RANK	Mi identificador de proceso
MPI_SEND	Envía un mensaje
MPI_RECV	Recibe un mensaje

Enlaces con Lenguajes

- Describe para un lenguaje base dado
  - Sintaxis concreta
  - Convenciones para el manejo de errores
  - Modos de los parámetros

- Lenguajes base populares:
  - C
  - Fortran

Ejemplo 1

```
#include <stdio.h>
#include <mpi.h>


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

  MPI_Init (&argc, &argv); /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size); /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

Funciones

```
int MPI_Init (&argc, &argv);
```

- Es una rutina de inicialización y debe ser llamada antes de cualquier otra rutina de MPI.
- Únicamente se debe llamar una vez.

```
int MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

- Un proceso es capaz de obtener su *identificador de proceso*

Funciones

```
int MPI_Comm_size (MPI_COMM_WORLD, &size);
```

- Determinar el número de procesos

```
int MPI_Finalize();
```

- Al hacer la llamada a *MPI_Finalize(void)* ya no se podrá hacer una llamada a cualquier función MPI
  - Ni siquiera *MPI_Init()*.
- El usuario debe cerciorarse que todos las comunicaciones pendientes que involucran a un proceso estén terminadas antes de que el proceso llame a la rutina MPI_Finalize().

Paso de mensajes Punto-a-Punto

- Mensajes enviados de un procesador a otro son ordenados como FIFO
- Los mensajes enviados por diferentes procesadores llegan de forma no determinística

Paso de mensajes Punto-a-Punto

- El receptor puede especificar la fuente
  - fuente = identidad del emisor => nombrado simétrico
  - fuente = MPI_ANY_SOURCE => nombrado asimétrico
  - ejemplo: especifica el emisor del siguiente renglón pivote en ASP

- El receptor también puede especificar una etiqueta
  - Distingue entre diferentes tipos de mensajes
  - Similar to operation name in SR or entry name in ADA

Funciones

```
int MPI_Send( void *buf, int count, MPI_Datatype
  datatype, int dest, int tag,MPI_Comm comm )
```

- Envia datos en un mensaje
- Los datos a enviar inician a partir del primer elemento indicado en buf,
- El número de elementos se indica en count
- El tipo de datos de cada elemento en buffer se indica en datatype
- identificador del destinatario se indica en dest
- Bandera del mensaje en tag
- El comunicador en comm
- El comunicador en comm. Es un comunicador predefinido que consiste de todos los procesos en ejecución cuando la ejecución del programa inicia

Funciones

```
int MPI_Recv( void *buf, int count, MPI_Datatype
  datatype, int source, int tag, MPI_Comm comm,
  MPI_Status *status )
```

- Recibir datos de un mensaje
  - Los datos recibidos se almacenan en buf
  - La dirección de una estructura que indica el estatus de la recepción
- Hay que indicar como parámetros
  - El número máximo de elementos a recibir está establecido en count
  - Tipo de datos de cada elemento en buffer se indica en datatype
  - Identificador del remitente en source
  - Bandera del mensaje en tag
  - El comunicador en comm. Es un comunicador predefinido que consiste de todos los procesos en ejecución cuando la ejecución del programa inicia

Ejemplos (1/2)

```
int x, status;
float buf[10];

MPI_SEND (buf, 10, MPI_FLOAT, 3, 0, MPI_COMM_WORLD);
   /* envía 10 floats al proceso 3 MPI_COMM_WORLD = todos
   los procesos */


MPI_RECV (&x, 1, MPI_INT, 15, 0, MPI_COMM_WORLD, &status);
   /* recibe 1 entero desde el proceso 15 */


MPI_RECV (&x, 1, MPI_INT, MPI_ANY_SOURCE, 0,
   MPI_COMM_WORLD, &status);
   /* recibe 1 entero desde cualquier proceso */
```

Ejemplos (2/2)

```
int x, status;
#define NEW_MINIMUM 1

MPI_SEND (&x, 1, MPI_INT, 3, NEW_MINIMUM, MPI_COMM_WORLD);
      /* send message with tag NEW_MINIMUM */.


MPI_RECV (&x, 1, MPI_INT, 15, NEW_MINIMUM, MPI_COMM_WORLD,
   &status);
   /* receive 1 integer with tag NEW_MINIMUM */


MPI_RECV (&x, 1, MPI_INT, MPI_ANY_SOURCE, NEW_MINIMUM,
   MPI_COMM_WORLD, &status);
   /* receive tagged message from any source */
```

Formas de pasar los mensajes (1)

- Modos de comunicación:
  - Standard
    - El sistema decide cuando el mensaje es almacenado en un buffer
  - Almacenado en buffer:
    - El usuario explícitamente controla el almacenamiento en el buffer
  - Síncrono:
    - El send espera a que haya un receive
  - Listo:
    - El send puede iniciar solo si se ha posteado un receive correspondiente

Formas de pasar los mensajes (2)

- Comunicación no-bloqueante
  - Cuando un send bloqueante regresa, el buffer de memoria puede ser reusado
  - Un receive bloqueante espera un mensaje
  - Un send no-bloqueante regresa inmediatamente (peligroso)
  - Un receive no-bloqueante a través de IPROBE

Receive no-bloqueante

MPI_IPROBE	Checar mensajes pendientes
MPI_PROBE	Espera un mensaje pendiente
MPI_GET_COUNT	Número de elementos de datos en el mensaje

MPI_PROBE (source, tag, comm, &status)	Status
MPI_GET_COUNT (status, datatype, &count)	Tamaño del mensaje
status.MPI_SOURCE	Identifica emisor
status.MPI_TAG	Etiqueta del mensaje

Ejemplo: Checar un mensaje pendiente

```
int buf[1], flag, source, minimum;
while ( ...) {
   MPI_IPROBE(MPI_ANY_SOURCE, NEW_MINIMUM, comm, &flag, &status);
   if (flag) {   // Si hay un mensaje
        /* handle new minimum */
        source = status.MPI_SOURCE;   // Determinar quien lo tiene
        MPI_RECV (buf, 1, MPI_INT, source, NEW_MINIMUM, comm,
   &status);
        minimum = buf[0];
   }
   ...  /* compute */
}
```

Ejemplo: Recibiendo un mensaje con un tamaño desconocido

```
int count, *buf, source;
MPI_PROBE(MPI_ANY_SOURCE, 0, comm, &status);
source = status.MPI_SOURCE;
MPI_GET_COUNT (status, MPI_INT, &count);
buf = malloc (count * sizeof (int));
MPI_RECV (buf, count, MPI_INT, source, 0, comm, &status);
```

Operaciones Globales – Comunicación Colectiva

- Comunicación coordinada entre todos los procesos
- Funciones:

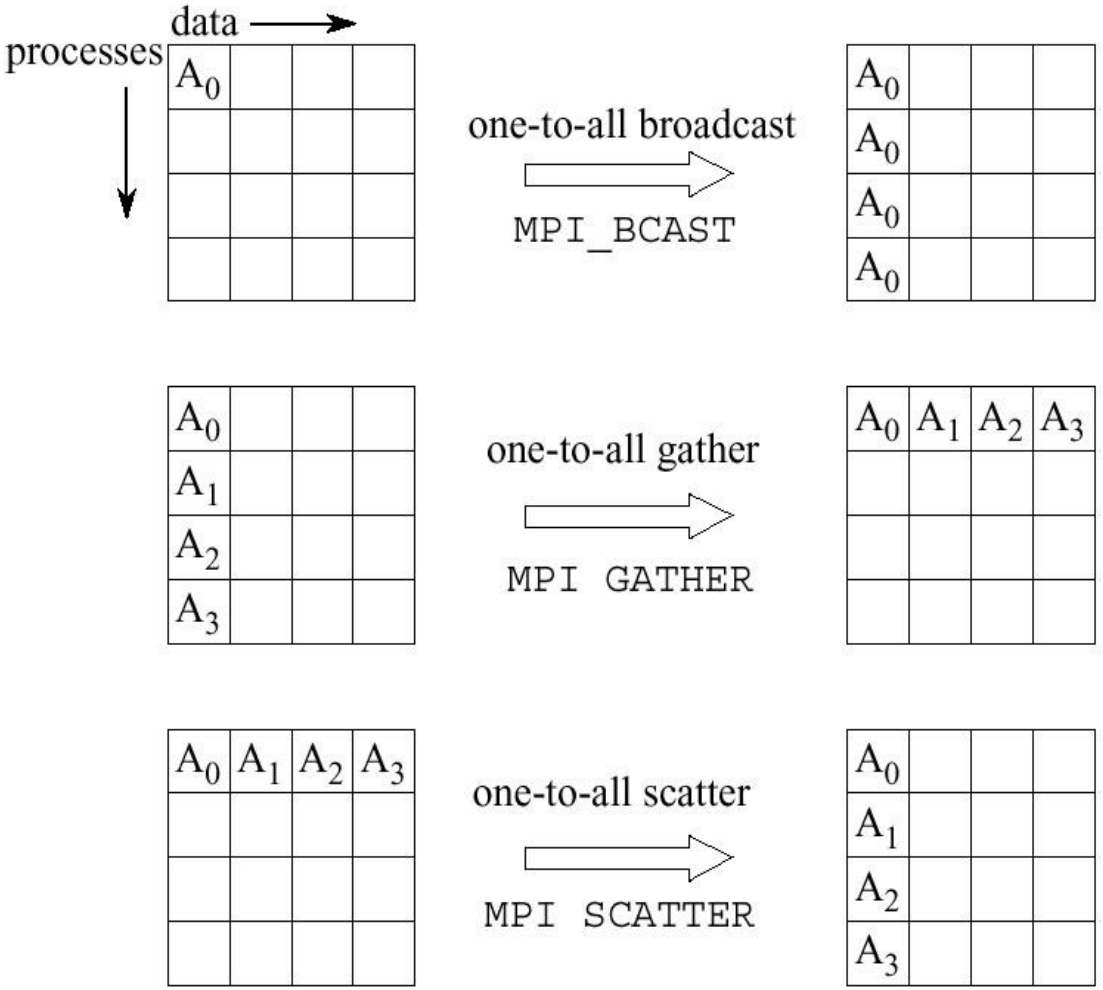
MPI_BARRIER	Sincroniza todos los procesos
MPI_BCAST	Envía datos a todos los procesos
MPI_GATHER	Recopilar datos de todos los procesos
MPI_SCATTER	Dispersar datos a todos los procesos
MPI_REDUCE	Operación de reducción
MPI_REDUCE ALL	Reducción, todos los procesos obtienen el resultado

Barreras

```
int MPI_Barrier (MPI_Comm comm )
```

- Sincroniza un grupo de procesos
- Todos los procesos se bloquean hasta que todos hayan llegado a la barrera
- Comunmente usado al final de un ciclo en algoritmos iterativos

Figura 8.3 del libro de Foster



Reducción

- Combina valores provistos de diferentes procesos
- Los resultados se envían a un procesador (MPI REDUCE) o a todos los procesadores (MPI REDUCE ALL)
- Usado con operaciones conmutativas y asociativas
  - MAX, MIN, +, x , AND, OR

Ejemplo 1

- Operación global mínima

```
MPI_REDUCE (inbuf, outbuf, 2, MPI_INT, MPI_MIN, 0,
   MPI_COMM_WORLD)
```

- `outbuf[0]` = mínimo entre los `inbuf[0]` de todos los procesos
- `outbuf[1]` = mínimo entre los `inbuf[1]` de todos los procesos

Figura 8.4 del libro de Foster

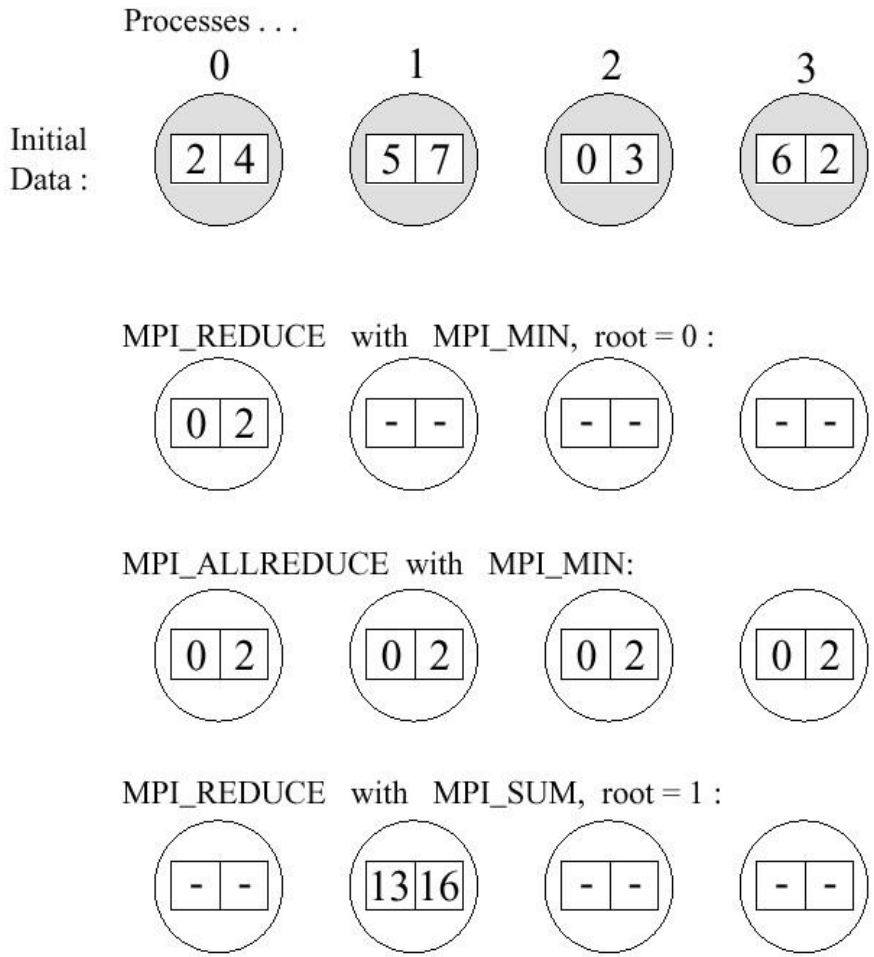
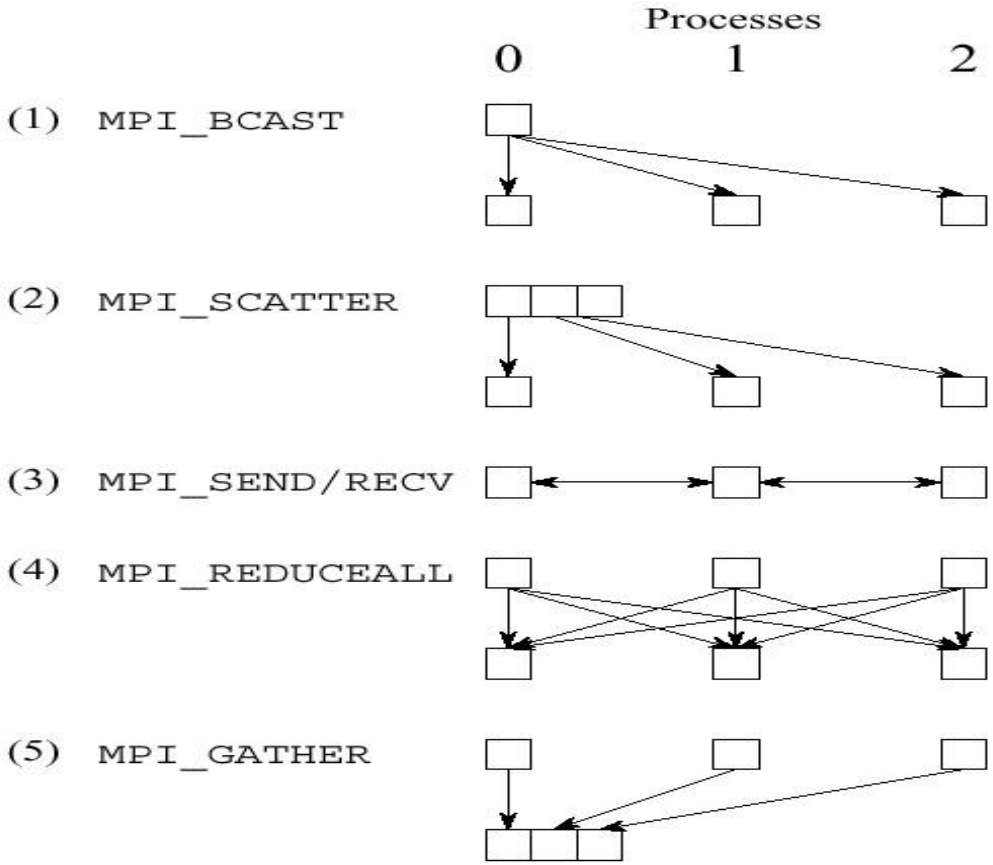


Figura 8.5 del libro de Foster



Modularidad

- Los programas en MPI usan librerías
- Las funciones de librería pueden enviar mensajes
- Estos mensajes no deben interferir con los mensajes de la aplicación
- Las etiquetas (tags) no resuelven ese problema

Comunicadores

- Un comunicador denota un grupo de procesos (contexto)
- MPI_SEND y MPI_RECV especifican un comunicador
- MPI_RECV solo puede recibir mensajes enviados al mismo comunicador
- Las funciones de librería pueden usar comunicadores separados, enviados como parámetro

Discusión

- Basado en librería:
  - No se requieren modificaciones al lenguaje
- La sintaxis es compleja
- La recepción del mensaje está basada en la identidad del emisor y etiqueta de la operación, pero no en el contenido del mensaje

Sintaxis

```
SR:
   call slave.coordinates(2.4, 5.67);
   in coordinates (x, y);

MPI:
   #define COORDINATES_TAG 1
   #define SLAVE_ID 15
   float buf[2];
   buf[0] = 2.4; buf[1] = 5.67;
   MPI_SEND (buf, 2, MPI_FLOAT, SLAVE_ID, COORDINATES_TAG,
   MPI_COMM_WORLD);
   MPI_RECV (buf, 2, MPI_FLOAT, MPI_ANY_SOURCE,
   COORDINATES_TAG, MPI_COMM_WORLD, &status);
```