Comunicación punto a punto

Carlos E. Alvarez[1].

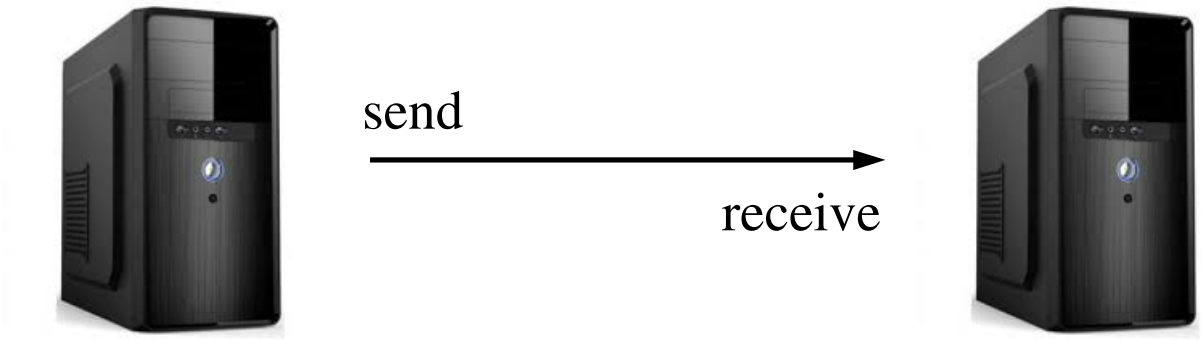
[1]Dep. de Matemáticas aplicadas y Ciencias de la Computación, Universidad del Rosario

2020-I

Componentes de un mensaje

- ID del proceso que envía el mensaje (*source*)
- ID del proceso al que está destinado el mensaje (*destination*)
- Tipo de datos que se envían (*datatype*)
- Cantidad de datos que se envía (*count*)
- Apuntador a los datos que se envían (*address*)
- ID del mensaje (*tag*)

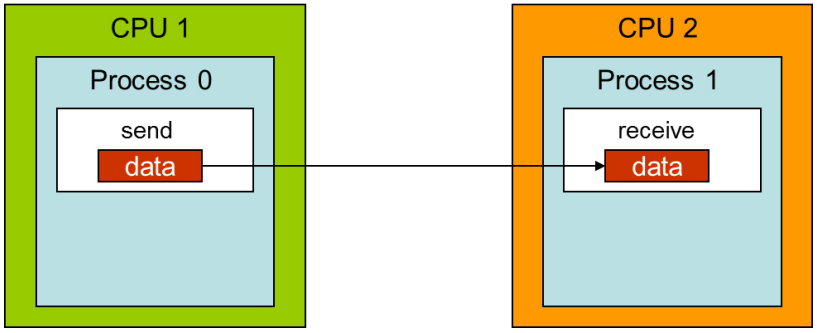
Comunicación punto a punto



Comunicación punto a punto.

Comunicación punto a punto

- Forma mas básica de comunicación
- Empareja una función `send` con su correspondiente `receive`
- Involucra solo dos procesos:
  - Fuente (*source*)
  - Destinatario (*destination*)



Fuente: Proceso 0. Destinatario: Proceso 1.

Función send

```
MPI_Send(address,count,datatype,destination,tag,comm)
```

*El valor mas común para `comm` es `MPI_COMM_WORLD`.

Función `receive`

```
MPI_Recv(address,maxcount,datatype,source,tag,comm,status)
```

- El segundo parámentro permite recibir mensajes de $\leq$ `maxcount` unidades
- `tag` debe emparejar con el de `MPI_Send`, a menos que se use un comodín (wildcard)
- dado que `maxcount` y `tag` pueden tener un margen de valores que reciven, `status` guarda los valores precisos de tamaño, etiqueta y fuente

Comunicador y etiqueta

- **Comunicador:** Objeto que define el grupo de procesos que participan en las comunicaciones
- **Etiqueta (tag):** Entero arbitrario que identifica un mensaje particular. La etiqueta que define la fuente debe coincidir con la que espera el destinatario

Para que una comunicación sea exitosa:

- La fuente debe especificar un rango de destino válido
- El destinatario debe especificar un rango de fuente válido
- Ambos procesos debe estar en el mismo comunicador
- Las etiquetas deben coincidir
- El buffer de datos del destinatario debe tener tamaño suficiente para guardar el mensaje

Comodines (wildcards):

- Para recivir mensajes de cualquier fuente: `MPI_ANY_SOURCE`
- Para recivir mensajes con cualquier etiqueta: `MPI_ANY_TAG`

Tipos de datos primitivos en MPI

Proveen compatibilidad entre diferentes tipos de máquinas.

C	**MPI**
char	MPI_CHAR
short	MPI_SHORT
int	MPI_INT
float	MPI_FLOAT
double	MPI_DOUBLE

Ejemplo

```
int main(int argc, char** argv){
  //initialize mpi environment
  MPI_Init(&argc, &argv);
  //Find out rank, size
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  //Assume at least 2 processes
  if(size < 2){
    printf("World size must be greater than 1\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  .
  .
  .
```

```
  .
  .
  .
  int number;
  if(rank == 0){
    //set number to -1 and send it to process 1
    number = -1;
    MPI_Send(&number,1,MPI_INT,1,0,MPI_COMM_WORLD);
  }else if(rank == 1){
    MPI_Recv(&number,1,MPI_INT,0,0,MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
             number);
  }

  MPI_Finalize();
  return 0;
}
```

Ejercicio: Escriba un programa que genere dos procesos y declare la variable

```
1       int pingpong_count = 0;
```

El primer proceso incrementa la variable en uno y pasa el valor al segundo proceso, quien incrementa de nuevo el valor en uno y lo regresa al primero, quien ingrementa el valor en uno y... . El programa debe realizar esta acción de ping-pong entre los dos procesos N veces. Para verificar la correctitud del programa, cada proceso imprime su identidad, la identidad del proceso a quien envía el valor y el valor que envía.

Preservación del orden de los mensajes

- Los mensajes llegan en el mismo orden en que se envían
- Si se usa un envío sincrónico pueden presentar problemas, si se reciben los mensajes en un orden distinto al que son enviados

Función `Ssend`

En la comunicación sincrónica la fuente bloquea su ejecución hasta que reciba una señal confirmando que su mensaje fué recibido por el destinatario

```
MPI_Ssend(address,count,datatype,destination,tag,comm)
```

Ejemplo

```
int main(int argc, char** argv){
  //Initialize MPI environment
  MPI_Init(&argc, &argv);
  //Find out rank, size
  int rank, size;
  MPI_Comm comm = MPI_COMM_WORLD; //Communicator
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);

  //Need at least 2 processes
  if(size < 2){
    printf("World size must be greater than 1\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  .
  .
  .
```

Ejemplo

```
.
int message1;
char message2;
if(rank == 0){
  //Initialize messages
  message1 = 5;
  message2 = 'e';
  //tags
  int tag1 = 1;
  int tag2 = 2;
  //receiver
  int other = 1;

  //Send fist message
  printf("Sending message 1...\n");
  MPI_Ssend(&message1, 1, MPI_INT, other, tag1, comm);
  //Send second message
  printf("Sending message 2...\n");
  MPI_Ssend(&message2, 1, MPI_CHAR, other, tag2, comm);
  .
```

Ejemplo

```
    .
  }else if(rank == 1){
    //tags
    int tag1 = 1;
    int tag2 = 2;
    //sender
    int other = 0;

    //Receive first message
    MPI_Recv(&message1, 1, MPI_INT, other, tag1, comm, MPI_STATUS
    printf("I received message 1: %d\n",message1);
    //Receive second message
    MPI_Recv(&message2, 1, MPI_CHAR, other, tag2, comm, MPI_STATU
    printf("I received message 2: %c\n",message2);
  }
  //Exit the mpi environment ------------------------------------
  MPI_Finalize();

  return 0;
}
```

Cambie el orden en el que se reciben los mensajes ¿Que sucede?

Ejercicio 1