Puzle, Parte 1

Mikel Dalmau

12 de Marzo de 2016

1 Comunicaciones colectivas

En la comunicación entre un grupo de individuos, existen mecanismos que permiten una comunicación más eficiente. En este trabajo en el apartado 1.1 *Comunicación en forma de árbol* se muestran varios modelos de comunicación compuestos por árboles.

En MPI, la comunicación colectiva consiste en una serie de funciones que sirven para que un grupo de procesos se comuniquen entre ellos. En el apartado 1.2 *Comunicaciones colectivas frente a comunicaciones Punto a Punto* se muestran las diferencias principales entre las funciones de comunicación colectiva y las de punto a punto. En el apartado 1.3 *Funciones de Comunicación Colectiva en MPI* se describen las funciones más utilizadas y sus parámetros.

1.1 Comunicación en forma de árbol

En [1] se describe esta forma de comunicación, mostrando como ejemplo un problema de suma *one-to-all*. Aunque inicialmente pueda parecer que no mejora demasiado, ya que la mitad de los nodos realizan las mismas comunicaciones que realizarían punto a punto(esto es, cuando todos los nodos envían su valor a un único nodo), mejora considerablemente reduciendo la cantidad de recepciones y sumas que tiene que realizar el nodo líder.

En el ejemplo de la imagen, el nodo 0 pasa de realizar *n-1* recepciones y sumas a realizar 3 recepciones y sumas, esto es, la altura del árbol $log_2\ n$($log_d\ n$ en el caso general, siendo d el grado de árbol). De esta forma la carga de trabajo de los nodos crece logarítmica-mente con el número de procesadores, la mejora es notable frente al crecimiento lineal de la comunicación centralizada.

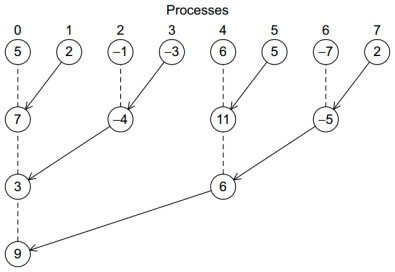

FIGURE 3.6
A tree-structured global sum

El mismo modelo invertido se puede utilizar para comunicación *one-to-all*.

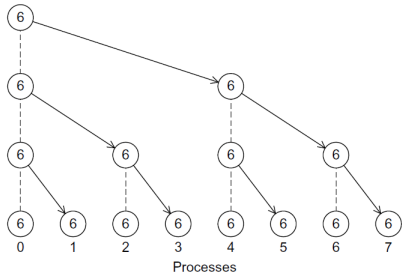

FIGURE 3.10
A tree-structured broadcast

Este último ejemplo muestra un modelo de comunicación all-to-all en el que se construyen n árboles siendo cada nodo raíz de uno de ellos.

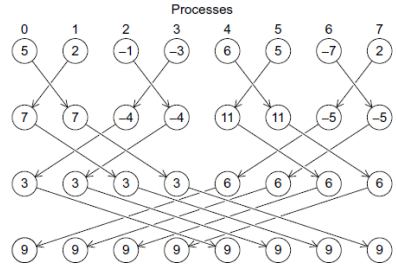

FIGURE 3.9
A butterfly-structured global sum

Año: 2015 - 2016
Profesor(a): Jesus Bermudez De Andres
Teléfono:
E-mail: mdalmau002@ikasle.ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

El problema de este modelo de comunicación es la dificultad de programarlo, existen incontables maneras distintas de hacerlo pero no sabríamos cual es la mejor, para que tipos y tamaños de problemas cual es la óptima. En [2] se sugiere también el uso de árboles de profundidad logarítmica para la comunicación, pero tampoco indican o desconocen si las funciones MPI hacen uso de ellos.

1.2 Comunicaciones colectivas frente a comunicaciones Punto a Punto

1. Todos los procesos en el comunicador deben llamar a la misma función colectiva independientemente del tipo de comunicación que se realice. Mientras en la comunicación punto a punto queda definido por la llamada quien es emisor y quien es receptor, MPI_Recv y MPI_Send.
2. Todos los procesos en el comunicador tienen que tener los parámetros compatibles, esto es, si en una llamada MPI_Reduce dos procesos tiene *root* distinto el programa va a fallar.
3. Las funciones de comunicación punto a punto se conectan mediante etiquetas (*tags*) y comunicadores, en las colectivas solamente mediante comunicadores y el orden en el que son llamadas.
4. Otra restricción que las funciones de comunicación punto a punto no tienen es que la cantidad de datos enviados en el bufer tiene que coincidir exactamente con la esperada en el destino.
5. No exiten funciones de comunicación colectivas que no sean bloqueante, todas utilizan la función Barrier para asegurar las sincronización de los datos.

1.3 Funciones de Comunicación Colectiva en MPI

Podemos dividir las funciones de comunicación colectiva en función del tipo de comunicación que realizan y también en función del tipo de dato que manejan ya que así las distingue MPI.

- *all-to-one*:
  - MPI_REDUCE
  - MPI_GATHER
- *one-to-all*
  - MPI_BCAST
  - MPI_SCATTER
- *all-to-all*
  - MPI_ALLREDUCE
  - MPI_ALLGATHER
  - MPI_ALLTOALL
  - MPI_REDUCE_SCATTER
- *all-to-some*
  - MPI_SCAN

Respecto al tipo de dato que manejan, MPI dispone de las *vector variant* de las funciones vistas; MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV.

Estas funciones se diferencian por...

Respecto a las funciones ALL, en estas los parámetros son idénticos que los de sus semejantes con la excepción de que sobra el parámetro *root* ya que todos recibirán el mensaje.

1.3.1 MPI_Bcast

```
NAME
     MPI_Bcast -  Broadcasts a message from the
     process with rank "root" to all other processes
     of the communicator

SYNOPSIS
     int MPI_Bcast( void *buffer, int count,
     MPI_Datatype datatype, int root,
            MPI_Comm comm )
```

1.3.2 MPI_Reduce

```
NAME
     MPI_Reduce -  Reduces values on all
            processes to a single value

SYNOPSIS
     int MPI_Reduce(MPICH2_CONST void *sendbuf,
     void *recvbuf, int count, MPI_Datatype
     datatype, MPI_Op op, int root, MPI_Comm comm)
```

Es de especial interés el parámetro *op* que define el tipo de operación a realizar. Existen los siguientes tipos de operaciones:

Operation Value	Meaning
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
MPI_LAND	And lógico
MPI_BAND	And binario
MPI_LOR	Or lógico
MPI_BOR	Or binario
MPI_LXOR	Or exclusivo lógico
MPI_BXOR	Or exclusivo binario
MPI_MAXLOC	Máximo y su dirección
MPI_MINLOC	Mínimo y su dirección

1.3.3 MPI_Scatter

1.3.4 MPI_Gather

2 Ejercicios

2.1 P1.1

Hay que repartir un vector de N elementos entre npr procesos. Completa el programa serie *P11-distribute0.c*, para que genere el tamaño de cada trozo del vector y el desplazamiento desde el origen del vector al comienzo de cada trozo, en estos dos casos:

a. los posibles restos se añaden al último trozo

b. los posibles restos se añaden uno a uno a diferentes trozos

---

a. Inicialmente calculo Nloc y remainder.

```
//Compute Nloc and remainder
Nloc = floor((double)N/(double)npr);
remainder = N - Nloc*npr;
```

Este caso es sencillo y se resuelve con el siguiente bucle y las asignaciones finales.

```
//We distribute the work among the
    procesesses
for(i=0; i<npr-1; i++){
    size[i] = Nloc;
    shift[i] = i*Nloc;
}
//Finally we charge the last process with
    the remainder
size[npr-1] = Nloc + remainder;
shift[npr-1] = (npr-1)*Nloc;
```

b. En este segundo caso he comenzado distribuyendo la carga del resto entre los primeros procesadores, luego, las cargas distintas entre procesos no permiten el cálculo de shift usado anteriormente, por lo que tomo las referencias de tamaño y shift calculadas en la anterior iteración para calcular el shift, sabemos donde empezamos porque sabemos dónde termina el anterior.

```
//Value asignment to first process
size2[0] = Nloc;
shift2[0] = 0;

//Distribution of the remainder among the
    first processes
i = 0;
while(remainder){
    size2[i] += 1;
    remainder -= 1;
    i++;
}

//Distribute the rest of the vector among
    the processes
for(i=1; i<npr; i++){
    size2[i] += Nloc;
    shift2[i] = shift2[(i-1)] + size2[(i-1)
        ];
}
```

2.2 P1.2

El programa *P12-inteser.c* calcula el valor de una integral mediante el conocido método de sumar las áreas de n trapecios bajo la curva que representa una función. A mayor valor de n, más preciso el resultado.

Completa el programa MPI *P12-inteser.c* para realizar esa misma función entre P procesos, utilizando funciones de comunicación colectiva. Compara el resultado con el de la versión serie.

Para resolver el problema es necesario modificar la función Read_data, encargada de leer por pantalla los límites superior, inferior y número de evaluaciones de la función. Dado que todos los nodos no pueden utilizar la entrada estándar al mismo tiempo, limito la lectura a un único nodo y luego distribuyo los datos leídos entre el resto de procesos utilizando la función **MPI_Bcast**.

```
void Read_data(double* a_ptr, double* b_ptr,
     int* n_ptr, int pid){

  float a, b;
  float buf[3];

  if (pid == 0){
    printf("\n  Introduce a, b (limits) and
        n (num. of trap.)  \n");
    scanf("%f %f %d", &a, &b, n_ptr);
    buf[0] = a;
    buf[1] = b;
    buf[2] = (float)*n_ptr;
  }
  //Distribute read values
  MPI_Bcast(&buf,3,MPI_INT,0,MPI_COMM_WORLD)
      ;

  (*a_ptr)= (double)(buf[0]);
  (*b_ptr)= (double)(buf[1]);
  (*n_ptr)= (double)(buf[2]);
```

Tras realizar el cálculo del área correspondiente en cada proceso, es necesario sumarlas todas para obtener la integral en todo el intervalo. Esto puede realizarse en una única línea llamando a la función **MPI_Reduce** con los siguientes parámetros.

```
/*
```

```
Adding the partial results,

  Description of parameters:
  sendbuf  - local result of integral
  recvbuf  - total result of integral
  count    - 1 element in send buffer per
      porcess
  datatype - We are using double precission
  op       - We will compute a sume
  root     - the process 0
  comm     - All the processes active
*/
 MPI_Reduce(&resul_loc, &resul, 1,
     MPI_DOUBLE   , MPI_SUM, 0,
     MPI_COMM_WORLD);
```

Bibliografía

[1] Pacheco P.:*An Introduction to Parallel Programming*. Morgan Kaufmann,2011. Capítulo 3, apartado 4.

[2] Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J. *MPI: The Complete Reference, Volume 1, The MPI Core*. The MIT PRess,1999. Capítulo 4.