Arquitectura de Computadores

- Pablo Rayón Zapater
- Fernando Pérez Ballesteros
- José María Fernández

Práctica 4

UNIVERSIDAD NEBRIJA

Índice

Ejercicio 1

Crear un ejemplo sencillo de uso de cada primitiva explicada en esta práctica (MPI_Reduce, MPI_Allgather y MPI_Alltoall) y utilizarlo para explicar el funcionamiento de cada una de ellas. Estos ejemplos deben funcionar correctamente con cualquier número de procesos.

```c
int main(int argc, char **argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int array[size];
    int redSum[size];
    int allToall[size];
    int allGather[size];
    srand(time(NULL) + rank);
    int randAllGath = rand() % 10;

    printf("Sending number %d to process : %d through Allgather\n", randAllGath, rank);

    for (int i = 0; i < size; i++)
    {
        array[i] = i;
    }
    MPI_Reduce(&array, &redSum, size, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
    {
        for (int i = 0; i < size; i++)
        {
            printf("Reduce from process %d: %d\n", i, redSum[i]);
        }
    }
    MPI_Allgather(&randAllGath, 1, MPI_INT, &redSum, 1, MPI_INT, MPI_COMM_WORLD);
    if (rank == 0)
    {      Josojmf, hace 4 días • PR4 terminada …
        for (int i = 0; i < size; i++)
        {
            printf("Allgather from process %d: %d\n", i, redSum[i]);
        }
    }
    MPI_Alltoall(&array, 1, MPI_INT, &allToall, 1, MPI_INT, MPI_COMM_WORLD);

    for (int i = 0; i < size; i++)
    {
        printf("Alltoall from process %d: %d\n", rank, allToall[i]);
    }
    MPI_Finalize();
}
```

```
Sending number 6 to process : 1 through Allgather
Sending number 6 to process : 0 through Allgather
Sending number 8 to process : 2 through Allgather
Reduce from process 0: 0
Reduce from process 1: 3
Reduce from process 2: 6
Allgather from process 0: 6
Allgather from process 1: 6
Allgather from process 2: 8
Alltoall from process 0: 0
Alltoall from process 0: 0
Alltoall from process 0: 0
Alltoall from process 1: 1
Alltoall from process 1: 1
Alltoall from process 1: 1
Alltoall from process 2: 2
Alltoall from process 2: 2
Alltoall from process 2: 2
```

En este apartado se ha programado un código que ejecuta las tres funciones de comunicación colectiva que se piden en el enunciado. Las funciones Reduce y AlltoAll funcionan con un array que se inicializa en el primer bucle for con valores correspondientes a cada índice de este array ({1,2,3,4,5…n_procesos}). La función Allgather recibe un solo elemento de cada proceso y lo almacena en un array, este número es aleatorio y emplea como semilla el tiempo actual y el rango del proceso que lo ejecuta.

La función Reduce realiza una suma de sub-arrays que recibe de cada uno de los procesos, y en cada posición del array final almacena el resultado de esta suma, tomando como sumando el numero que cada proceso le envía, este correspondiendo a la posición del array final en la que se va a almacenar (para la posición 0 del array final todos los procesos envían su posición 0 y se suman). La función AllToAll envía a todos los procesos, incluido el mismo proceso que la ejecuta, una posición de su a array, al array estar inicializado con una serie secuencial ascendente comenzando en 0 (0,1,2,3…) cada proceso recibirá una cantidad de números igual al numero de procesos que se haya elegido, pero estos números serán iguales al rango de dicho proceso (el proceso 0 recibirá n_procesos x 0, el proceso 1 n_procesos x 1…).

Ejercicio 2

Implementar un programa que realice la transposición de la matriz inicial mostrada en la figura (parte izquierda). Realizar el proceso en paralelo, distribuyendo la matriz entre 4 procesos.

```
joso@DESKTOP-1M4B0KO:~/192.168.1.181:8000/PRACTICA_4$ mpirun -np 4 ./exec
====================== Original Matrix ======================
Row 0: 1 2 3 4
Row 1: 5 6 7 8
Row 3: 13 14 15 16
Row 2: 9 10 11 12
====================== Transposed Matrix ======================
Row 0: 1 5 9 13
Row 1: 2 6 10 14
Row 2: 3 7 11 15
Row 3: 4 8 12 16
```

```c
int rank, size;
struct timespec ts;                 // Time struct for nanosleep
ts.tv_sec = 100 / 1000;             // Time to sleep in seconds
ts.tv_nsec = (100 % 1000) * 1000000; // Time to sleep in nanoseconds

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int send[4], recv[4];
if (size == 4)
{
    if (rank == 0)
    {
        printf("====================== Original Matrix ======================\n");
    }
    nanosleep(&ts, &ts);
    printf("Row %d: ", rank);
    for (int i = 0; i < size; i++)
    {
        send[i] = (i + 1) + rank * size;
        printf("%d ", send[i]);
    }
    printf("\n");
    nanosleep(&ts, &ts); // Sleep for 100ms to divide the output of the processes
    if (rank == 0)
    {
        printf("====================== Transposed Matrix ======================\n");
    }
    MPI_Alltoall(&send, 1, MPI_FLOAT, &recv, 1, MPI_INT, MPI_COMM_WORLD);
    printf("Row %d: ", rank);
    for (int i = 0; i < size; i++)
    {
        printf("%d ", recv[i]);
    }
    printf("\n");
}
else
{
    printf("Error formato en los procesos\n");
}

MPI_Finalize();
```

A la derecha se muestra el código que realiza la tarea requerida, a la izquierda el output de este programa.

El programa realiza la transposición de una matriz por filas. Primero comienza comprobando que el numero de procesos que va a ejecutar esta tarea sea 4, ya que cada proceso se va a encargar de tratar una fila por separado y transponerla a través de la función AllToAll. La matriz se inicializa con un bucle for el cual asigna a cada posición de la matriz el numero de su índice, como la matriz es tratada como 4 arrays diferentes, siendo una fila para cada proceso, es necesario actualizar el valor de ese dato que se guarda en el array de cada proceso con el propio rango del proceso, así cada sub-array tendrá un offset que variará con cada proceso (sub-array 1: {1,2,3,4}, sub-array 2: {5,6,7,8} que corresponde al número de la fila multiplicado por el número de procesos, pudiendo comenzar así cada fila donde finalizó la anterior).
Para realizar la transposición de la matriz se ejecuta la función AlltoAll, la cual envía el primer número de cada array a la primera posición del array todos los procesos, este primer número de cada array si es colocado secuencialmente se convierte en la primera columna de la matriz, solo que esta vez puesta en forma de fila. Como la función AllToAll hace esto mismo con todas las posiciones de cada sub-array, acabamos consiguiendo que se coloquen por filas cada columna de la matriz original, dando lugar así a la transposición de esta.
Por último, mencionar que se ha implementado una función llamada nanosleep que permite a cada proceso dormir en función del número de nanosegundos que se le indique, se ha hecho esto ya que con la función sleep normal la ejecución era muy lenta ya que es imposible dormir un proceso por debajo de un segundo. Se ha dormido a los procesos con fin de mejorar el output del programa y poder observar separadamente la matriz original y la matriz transpuesta.
Referencia tomada para la función: https://stackoverflow.com/a/1157217.

Ejercicio 3

Conversión a decimal un número binario. Haciendo uso de las funciones explicadas en este guion, implementar un programa donde los procesos calculan en paralelo la representación decimal de un número binario:
• El número de procesos con el que se lance la aplicación será igual que el número de bits del número binario. 5
• Al iniciar la aplicación cada proceso genera un valor aleatorio (0 o 1) y se lo envía al resto. Uno de los procesos deberá imprimir por pantalla el número que se va a pasar a decimal.
• Una vez que todos los procesos tienen el dato, hacen las operaciones necesarias para calcular el valor decimal de forma paralela.
• Solo un proceso conoce el resultado final en decimal y lo imprime por pantalla.

```c
int main(int argc, char **argv)
{

    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);       Josojmf, hace 4 días • Test …
    srand(time(NULL) + rank);

    int val = rand() % 2;
    int val2 = val * pow(2, rank);
    int recv = 0;
    int buff_recv[size];
    int buff_recv_bin[size];
    MPI_Allgather(&val2, 1, MPI_INT, &buff_recv, 1, MPI_INT, MPI_COMM_WORLD);
    MPI_Allgather(&val, 1, MPI_INT, &buff_recv_bin, 1, MPI_INT, MPI_COMM_WORLD);
    MPI_Reduce(&val2, &recv, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
    {
        for (int i = 0; i < size; i++)
        {
            printf("Bit from process %d: %d\n", i, buff_recv_bin[i]);
        }
        printf("Binary number: ");
        for (int i = size - 1; i >= 0; i--)
        {
            printf("%d", buff_recv_bin[i]);
        }
        printf("\nDecimal number: %d\n", recv);
    }
    MPI_Finalize();
```

```
Bit from process 0: 0
Bit from process 1: 0
Bit from process 2: 1
Bit from process 3: 0
Binary number: 0100
Decimal number: 4
```

Para este ejercicio es necesario compilar con la flag -lm para poder enlazar la librería que contiene la función pow e.g.: “mpicc Ejercicio3.c -o ejecutable -lm”

A la izquierda se muestra el código del ejercicio y a la derecha el resultado y salida por pantalla. Primero se ha comenzado generando un valor aleatorio entre 0 y 1, para poder generar así un bit aleatorio para cada proceso, después cada proceso realiza una conversión a binario de dicho bit, tomando en cuenta la posición que ocupará en el resultado final del numero binario generado, esto se hace tomando el rango del proceso como la posición del numero binario generado, tomando el proceso 0 como LSB. Después hacemos dos Allgathers, uno que envía el valor decimal calculado por cada proceso para poder realizar así el último cálculo que de como resultado el numero en decimal, y el otro simplemente se realiza para que se pueda mostrar por pantalla el bit generado por cada proceso y el numero binario que se va a formatear. Este último cálculo lo realizará únicamente un proceso, en este caso el que tiene rango 0, esto se consigue a través de una suma por medio de un reduce, en el que todos los procesos envían su valor calculado y el proceso 0 aplica una suma, dando, así como resultado un valor decimal. Se han realizado dos bucles for por motivos estéticos y para mejorar la compresión del ejercicio, este par de bucles se encargan de imprimir el bit generado por cada proceso y el numero binario final, se han tenido que separar estos bucles por cuestiones de formato, ya que el primero imprime los bits en orden de generación, comenzando por el 0 y se ha tomado “Big endian” como formato del número binario , y el bucle previo imprime el LSB como primer bit, se sabe que esto no es óptimo y es pesado a nivel de procesamiento, para mejorar el rendimiento del programa se propone eliminar uno de los dos bucles o incluso los dos y solo mostrar el resultado final por pantalla.