Plataformas de soporte computacional: Programación con MPI

Diego R. Llanos, Belén Palop
Departamento de Informática
Universidad de Valladolid
{diego,b.palop}@infor.uva.es

Índice

1. Paso de mensajes

El paradigma de paso de mensajes

- Se utiliza para ejecutar aplicaciones en paralelo cuando los computadores no comparten un espacio direccionable.
- En lugar de utilizar variables compartidas, se utilizan mensajes explícitos para las comunicaciones.
- Ventajas e inconvenientes: los mensajes explícitos se detectan fácilmente en el código, pero su ejecución suele ser lenta debido al coste de las comunicaciones.

El modelo MPI

- Es el modelo de paso de mensajes más popular.
- Existe una implementación de dominio público llamada MPICH2.
- Utilizaremos esa implementación para hacer algunas pruebas en local (sin usar nuestra máquina de memoria compartida, Geopar).

Lanzamiento de tareas MPI

- El entorno de MPI lanza tantas copias del proceso como sean necesarias, bien en la máquina local o de forma remota.
- El programa se escribe teniendo en mente que habrá N copias trabajando simultáneamente. Esto supone un cambio de mentalidad respecto de OpenMP (donde un único proceso se encarga de lanzar y de recoger los threads).

- El programa se compila con la orden `mpicc -o prog prog.c`.
- Para ejecutarlo, el comando es `mpirun -np P ./prog`. Hay que indicar el número total `P` de procesos a lanzar.

2. Un algoritmo secuencial

Un problema perfectamente paralelizable: satisfacibilidad de circuitos digitales

- Problema: dado un circuito digital con N entradas y una salida, decidir si alguna combinación de sus entradas pone su salida a 1.
- Algoritmo con complejidad $O(2^n)$: de tipo NP (hay que probar todas las combinaciones posibles de sus entradas).
- Sin embargo, el problema es perfectamente paralelizable, ya que la prueba de entradas puede hacerse en paralelo.
- Sea una función de 16 entradas como la que sigue:

$$S = (e_0 \vee e_1) \wedge (\overline{e_1} \vee \overline{e_3}) \wedge (e_2 \vee e_3) \wedge (\overline{e_3} \vee \overline{e_4}) \wedge (e_4 \vee e_5) \wedge (e_5 \vee \overline{e_6}) \wedge (e_5 \vee e_6) \wedge (e_6 \vee \overline{e_{15}}) \wedge (e_7 \vee \overline{e_8}) \wedge$$
$$(\overline{e_7} \vee \overline{e_{13}}) \wedge (e_8 \vee e_9) \wedge (e_8 \vee \overline{e_9}) \wedge (\overline{e_9} \vee \overline{e_{10}}) \wedge (e_9 \vee e_{11}) \wedge (e_{10} \vee e_{11}) \wedge (e_{12} \vee e_{13}) \wedge (e_{13} \vee \overline{e_{14}}) \wedge (e_{14} \vee e_{15})$$

Un algoritmo secuencial

```
#include <stdio.h>

int main (int argc, char *argv[]) {
   int i;
   void check_circuit (int);

   for (i=1; i<65536; i++)
      check_circuit (i);
   return 0;
}

#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)

void check_circuit (int z) {
   int v[16];
   int i;

   for (i=0;i<16; i++)
      v[i] = EXTRACT_BIT(z,i);

   if ( (v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
      && (!v[3] || !v[4]) && (v[4] || !v[5])
      && (v[5] || !v[6]) && (v[5] || v[6])
      && (v[6] || !v[15]) && (v[7] || !v[8])
      && (!v[7] || !v[13]) && (v[8] || v[9])
      && (v[8] || !v[9]) && (!v[9] || !v[10])
      && (v[9] || v[11]) && (v[10] || v[11])
      && (v[12] || v[13]) && (v[13] || !v[14])
      && (v[14] || v[15]) ) {
         printf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d\n",
            v[0], v[1], v[2], v[3], v[4], v[5], v[6], v[7], v[8],
            v[9], v[10], v[11], v[12], v[13], v[14], v[15]);
         fflush(stdout); }
}
```

Su versión paralela con MPI

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

int main (int argc, char *argv[]) {
   int i;
   int id;
   int p;
   void check_circuit (int, int);

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   for (i=id; i<65536; i+=p)
      check_circuit (id, i);

   printf("Process %d is done.\n", id);
   fflush (stdout);
   MPI_Finalize();
   return 0;
}

#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)

void check_circuit (int id, int z) {
   int v[16];
   int i;

   for (i=0;i<16; i++)
      v[i] = EXTRACT_BIT(z,i);

   if ( (v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
      && (!v[3] || !v[4]) && (v[4] || !v[5])
      && (v[5] || !v[6]) && (v[5] || v[6])
      && (v[6] || !v[15]) && (v[7] || !v[8])
      && (!v[7] || !v[13]) && (v[8] || v[9])
      && (v[8] || !v[9]) && (!v[9] || !v[10])
      && (v[9] || v[11]) && (v[10] || v[11])
      && (v[12] || v[13]) && (v[13] || !v[14])
      && (v[14] || v[15]) ) {
         printf("%d) %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d\n",id,
            v[0], v[1], v[2], v[3], v[4], v[5], v[6], v[7], v[8],
            v[9], v[10], v[11], v[12], v[13], v[14], v[15]);
         fflush(stdout); }
}
```

3. Descripción de funciones MPI

Descripción de funciones MPI

- `MPI_Init(&argc, &argv)`: Se encarga de inicializar el sistema MPI. Debe hacerse esta llamada antes de ejecutar ninguna otra función con MPI.
- `MPI_Comm_rank()` y `MPI_Comm_size()`: Al inicializar MPI, el proceso pasa a ser miembro de un "comunicador" llamado `MPI_COMM_WORLD`, un objeto que permite la comunicación entre procesos.

- Los procesos dentro de un comunicador están en orden. Ese orden es su **rango**, contando desde cero. `MPI_Comm_rank()` devuelve el rango del proceso actual (es decir, su identificador), mientras que `MPI_Comm_size()` devuelve el número total de procesos lanzados en ese rango.
- Finalmente, `MPI_Finalize()` "desconecta" a los procesos del motor MPI subyacente y libera los recursos asignados a aquéllos.
- En nuestro ejemplo, el procesamiento es simétrico: todos los procesos prueban prácticamente el mismo número de combinaciones.
- En general, habrá un proceso "maestro" que envíe a los "esclavos" la tarea a realizar, y esperará a que terminen.
- Para ello se usa `MPI_Comm_rank()`: si el identificador devuelto es, digamos, 0, entonces somos el proceso maestro, y un esclavo en caso contrario.
- La función `MPI_Reduce()` sirve para efectuar una operación de reducción entre varios procesos.
- Formato:

```
int MPI_Reduce(
   void *operando; /* dirección del operando a reducir */
   void *result;  /* dirección del resultado final */
   int count; /* reducciones a realizar */
   MPI_datatype; /* tipo del operando a reducir */
   MPI_Op; /* operador de reducción a aplicar */
   int root; /* identificador del proceso que usará el resultado */
   MPI_Comm comm) /* comunicador */
```

Ejercicios

1. Escribir y probar el ejemplo de "satisfacibilidad de circuitos" con MPI visto anteriormente.
2. Comprobar que funciona correctamente (nueve combinaciones lo satisfacen).
3. Modificar la función `check_circuit()` para que devuelva un 1 si la combinación satisface el circuito, y 0 en caso contrario.
4. Utilizar el valor devuelto por `check_circuit()` para que el proceso maestro muestre por pantalla el número total de combinaciones que satisfacen el circuito (a través de una operación de reducción).