
Paralelización del algoritmo de ordenamiento radixsort usando MPI y OpenMP

ARQUITECTURA E ENSEÑARÍA DE COMPUTADORES
CURSO 2012/2013

Penas Sabín, Darío <dario.penas@udc.es>
Pereira Guerra, Adrián <adrian.pereira@udc.es>
<https://github.com/adrisons/AEC>

Índice

1. Introducción	2
1.1. Ejemplo del algoritmo	3
1.2. La complejidad del algoritmo	5
2. Paralelización con MPI	5
A. Código secuencial	7
B. main con MPI	7
C. Implementación radixsort con MPI	8

1. Introducción

Radix sort es un algoritmo de ordenación cuyo rendimiento temporal en el peor caso es de $\mathcal{O}(kN)$ y de memoria $\mathcal{O}(k + N)$

Su principal característica es la utilización de un array de 10 posiciones denominado *bucket*, inicializado en cada una de las iteraciones a 0 y que guarda en cada una de las posiciones del array el número de apariciones de cada una de las cifras coincidentes con la propia posición del *bucket*.

Este *bucket* es utilizado para calcular el propio *bucket* acumulado con el que asignaremos una posición diferente a cada uno de los elementos en el array en *b*.

En cada una de las iteraciones tenemos una variable llamada *exp* que se irá multiplicando por 10 y que nos indica la cifra que miraremos en ese instante y que luego copiaremos a *a*, que es el array inicial.

Este es el código de dicho algoritmo:

```
1 // Uso del bucket en radix sort
2 while (m / exp > 0){
3     int bucket[10] = {0};
4     for (i = 0; i < n; i++){
5         bucket[a[i] / exp % 10]++;
6     }
7     for (i = 1; i < 10; i++)
8         bucket[i] += bucket[i - 1];
9     for (i = n - 1; i >= 0; i--)
10        b[--bucket[a[i] / exp % 10]] = a[i];
11    for (i = 0; i < n; i++){
12        a[i] = b[i];
13    }
14    exp *= 10;
15 }
```

1.1. Ejemplo del algoritmo

Vector inicial: 25 57 48 37 12 92 86 33

Los elementos quedarían ordenados de la siguiente manera:

0:
1:
2: 12 92
3: 33
4:
5: 25
6: 86
7: 57 37
8: 48
9:

El bucket, en vez de ser lo descrito anteriormente, quedaría con el número de elementos asignados a cada posición:

0: 0
1: 0
2: 2
3: 1
4: 0
5: 1
6: 1
7: 2
8: 1
9: 0

El vector después de esta iteración sería: 12 92 33 25 86 57 37 48

En la siguiente iteración nos centramos en la segunda cifra de cada uno de los elementos:

0:
1: 12
2: 25
3: 33 37
4: 48
5: 57
6:
7:
8: 86
9: 92

En esta ocasión el bucket quedaría de la siguiente forma:

0: 0
1: 1
2: 1
3: 2
4: 1
5: 1
6: 0
7: 0
8: 1
9: 1

En este ejemplo el vector ya ha quedado ordenado: 12 25 33 37 48 57 86 92

1.2. La complejidad del algoritmo

2. Paralelización con MPI

Como este código es muy complicado de paralelizar, limitaremos el problema a dos procesadores, para simplificar el paso de datos.

Antes de comenzar a ordenar los datos es necesario calcular el máximo de todos los elementos, ya que influye en el número de ejecuciones del algoritmo, por lo tanto, es necesario que todos los procesos conozcan este dato. Todos los procesos tienen que ejecutar el mismo número de iteraciones, aunque puede que alguno no necesite hacer tantas, para sincronizar el paso de datos final, con el array ordenado. Esto se hace con `MPI_Allreduce`, que realiza la operación indicada (`MPI_MAX` en este caso) sobre el dato indicado (`m`) y comparte el resultado con **todos** los procesos.

```
1 m = maximo(a, n);  
2 MPI_Allreduce(&m, &m, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
```

El bucket se recalcula en cada iteración, por lo que también es necesario compartirlo en cada iteración. Radixsort ordena en función de la suma acumulada del bucket.

El trabajo de realizar la suma acumulativa lo hace el proceso 1 y le manda el resultado al 0. Es necesario hacerlo de este modo, ya que se utiliza el bucket acumulado para calcular las posiciones destino de los elementos del array `a`, ordenándolo en cada iteración.

El cálculo de las posiciones destino se hace desde el final al principio del array a ordenar (así lo requiere el algoritmo). Además, se resta el bucket para obtener la posición destino por lo que, al paralelizarlo, es necesario secuencializar esta parte de los procesos, de modo que sigan el mismo orden que el secuencial. Para ello, los elementos se envían en orden *inverso* de procesos (desde el proceso `n` al `n - 1`, del `n - 1` al `n - 2`, ...).

Es raro que, al dividir los datos a ordenar entre dos procesos, coincida que los datos del proceso 0 pertenezcan a la primera mitad del array a ordenar y los de proceso 1 a la segunda. Es decir, se va a tener que enviar algún dato de P0 a P1. Para esto, se van almacenando los datos a intercambiar en un *array de intercambio* y, al final de cada iteración, los procesos intercambian sus `a_intercambio` y guardan los datos recibidos.

```
1 // El P0 se queda esperando por los datos del P1  
2 if(myrank==0){  
3     MPI_Recv(a_intercambio, elemTot, MPI_INT, 1, 3, MPI_COMM_WORLD, &  
4         status);  
5     MPI_Recv(&bucket[0], 10, MPI_INT, 1, 4, MPI_COMM_WORLD, &status);  
6     for(i=0; i<elemTot; i++){  
7         if(a_intercambio[i]!=-1){  
8             // Se adapta la posicion del otro proceso  
9             pos_envio = i % n;  
10            b[pos_envio] = a_intercambio[i];  
11        }  
12    }  
13    inicializar_array(&a_intercambio[0], elemTot, -1);  
14 }
```

Este array tiene las dimensiones del array total. Se ha decidido implementarlo así, aunque pueda suponer una mayor carga de memoria, porque los procesos no siempre van a querer enviar datos en el mismo instante, lo que supondría que los procesos se quedarán esperando mutuamente y repercutiría mucho sobre el tiempo de ejecución.

`a_intercambio` tiene las dimensiones del array total porque los elementos a intercambiar ya se guardan en la posición destino del otro proceso. De este modo, al guardar los datos en `a_intercambio`, cada proceso tiene que adaptar la posición en la que se guarda al otro proceso.

Cuando el P1 acaba una iteración, envía el bucket y el `a_intercambio` al P0 y se queda esperando por el `a_intercambio` de P0.

```

1  if (myrank==1){
2      MPI_Send(a_intercambio, elemTot, MPI_INT, 0, 3, MPI_COMM_WORLD);
3      MPI_Send(&bucket[0], 10, MPI_INT, 0, 4, MPI_COMM_WORLD);
4      MPI_Recv(a_intercambio, elemTot, MPI_INT, 0, 5, MPI_COMM_WORLD, &
           status);
5
6      for (i=0; i<n; i++){
7          if (a_intercambio[i] != -1){
8              b[i] = a_intercambio[i];
9          }
10     }
11     inicializar_array(&a_intercambio[0], elemTot, -1);
12 }

```

Al final de cada iteración del bucle, cada proceso recibe el `a_intercambio` del otro proceso, lo guarda en su array `b` local y, finalmente, lo almacena en `a`.

A. Código secuencial

```
1 #include <math.h>
2 #include <assert.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/time.h>
6 #define MAX 100000000
7
8 void radixsor(int *a, int n){
9     int i, m = a[0], exp = 1;
10    int *b = malloc(MAX*sizeof(int)+1);
11
12
13    for (i = 0; i < n; i++){
14        if (a[i] > m){
15            m = a[i];
16        }
17    }
18    while (m / exp > 0){
19        int bucket[10] = {0};
20        for (i = 0; i < n; i++){
21            bucket[a[i] / exp % 10]++;
22        }
23        for (i = 1; i < 10; i++){
24            bucket[i] += bucket[i - 1];
25        }
26        for (i = n - 1; i >= 0; i--){
27            b[--bucket[a[i] / exp % 10]] = a[i];
28        }
29        for (i = 0; i < n; i++){
30            a[i] = b[i];
31        }
32        exp *= 10;
33    }
34
35
36    int main(int argc, char* argv[]){
37        int numElementos = atoi(argv[1]);
38        int i;
39        int *array = malloc(MAX*sizeof(int)+1);
40        static int n;
41
42        for (n = 0; n < numElementos; n++){
43            int r = rand() % 1000000;
44            array[n] = r;
45        }
46        radixsor (&array[0], numElementos);
47
48        return 0;
49    }
```

./src/secuencial.c

B. main con MPI

```
1 int main(int argc, char* argv[]){
2     int numElementos = atoi(argv[1]);
3     int i, p, myrank;
```

```

4  MPI_Status status;
5  struct timeval t0, t1, t;
6
7  if(argc < 2){
8      printf("Usage: mpirun -n numprocs ./radix numElementos \n");
9      exit(1);
10 }
11 /* Inicializacion de MPI*/
12 MPI_Init (&argc, &argv);
13 /* myrank will contain the rank of the process
14 MPI_COMM_WORLD is all processors together */
15 MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
16 /* p es el numero de procesos */
17 MPI_Comm_size (MPI_COMM_WORLD, &p);
18 /* n es el numero de elementos de cada proceso*/
19 int n = numElementos/p;
20 int *a = malloc(n*sizeof(int));
21 inicializar_a_creciente(&a[0], n, myrank);
22 /* Ejecuta el algoritmo de ordenamiento*/
23 radixsort_parallel (a, numElementos, p, myrank);
24
25 if (myrank == 0)
26     MPI_Recv(a, n, MPI_INT, 1, 123, MPI_COMM_WORLD, &status);
27 else
28     MPI_Send(a, n, MPI_INT, 0, 123, MPI_COMM_WORLD);
29
30 free(a);
31 MPI_Finalize ();
32 }

```

./src/main.c

C. Implementación radixsort con MPI

```

1 void radixsort_parallel(int* a, int elemTot, int p, int myrank){
2     int i, m, exp = 1;
3     struct timeval t0, t1, t;
4     int rank_send, rank_recv;
5     int n = elemTot/p;
6     int pos_envio = -1;
7     int* b = malloc(sizeof(int)*n);
8     int* a_intercambio = malloc(sizeof(int)*elemTot);
9     int* bucket = malloc(sizeof(int)*10);
10    int otro_rank = (myrank+1) % p;
11    int iter = 0;
12    MPI_Status status;
13
14    inicializar_array(&a_intercambio[0], elemTot, -1);
15    inicializar_array(&b[0], n, -1);
16    m = maximo(a, n);
17    MPI_Allreduce(&m, &m, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
18    while (m / exp > 0){
19        inicializar_array(bucket, 10, 0);
20        // Se calcula el bucket
21        for (i = 0; i < n; i++){
22            bucket[a[i] / exp % 10]++;
23        }
24        // Se comunican los procesadores para tener los mismos valores
        // en bucket

```



```

25 MPI_Allreduce(&bucket[0], &bucket[0], 10, MPI_INT, MPI_SUM,
26 MPI_COMM_WORLD);
27
28 if(myrank==1){
29     // Se calculan las frecuencias acumulativas de los elementos
30     // del bucket
31     for (i = 1; i < 10; i++)
32         bucket[i] += bucket[i - 1];
33 }
34 // El P0 se queda esperando por los datos del P1
35 if(myrank==0){
36     MPI_Recv(a_intercambio, elemTot, MPI_INT, 1, 3,
37 MPI_COMM_WORLD, &status);
38 MPI_Recv(&bucket[0], 10, MPI_INT, 1, 4, MPI_COMM_WORLD, &
39 status);
40 for(i=0;i<elemTot;i++){
41     if(a_intercambio[i]!=-1){
42         // Se adapta la posicion del otro proceso
43         pos_envio = i % n;
44         b[pos_envio] = a_intercambio[i];
45     }
46 }
47 inicializar_array(&a_intercambio[0], elemTot, -1);
48 }
49 // Usando el bucket se guardan en b los elementos de a
50 // ordenados
51 for (i = n - 1; i >= 0; i--){
52
53     int pos = (--bucket[a[i] / exp % 10]);
54
55     rank_recv = floor(pos / n);
56     rank_send = myrank;
57
58     if( rank_send == rank_recv ){
59         if(myrank==1)
60             pos = pos % n;
61         b[ pos ] = a[i];
62     }else{
63         if(myrank==0)
64             pos = pos % n;
65         a_intercambio [pos] = a[i];
66     }
67 }
68 // El P0 envia el array de intercambio al P1 para continuar
69 // ordenando
70 if(myrank==0){
71     MPI_Send(a_intercambio, elemTot, MPI_INT, 1, 5,
72 MPI_COMM_WORLD);
73 }
74 // El proceso 1 envia los elementos que le corresponden del
75 // proceso 0
76 if(myrank==1){
77     MPI_Send(a_intercambio, elemTot, MPI_INT, 0, 3,
78 MPI_COMM_WORLD);
79 MPI_Send(&bucket[0], 10, MPI_INT, 0, 4, MPI_COMM_WORLD);
80 MPI_Recv(a_intercambio, elemTot, MPI_INT, 0, 5,
81 MPI_COMM_WORLD, &status);
82
83 for(i=0;i<n;i++){
84     if(a_intercambio[i]!=-1){
85         b[i] = a_intercambio[i];
86     }
87 }

```

```

77     }
78     inicializar_array(&a_intercambio[0], elemTot, -1);
79 }
80 // Se guardan en a los elementos ordenados
81 for (i = 0; i < elemTot; i++){
82     if(b[i] != -1){
83         a[i] = b[i];
84     }
85 }
86 inicializar_array(&b[0], elemTot, -1);
87 exp *= 10;
88 iter++;
89 }
90 free(bucket);
91 }

```

./src/radixsort_mpi.c