# Paralelización del algoritmo de ordenamiento radixsort usando MPI y OpenMP

Arquitectura e Enxeñaría de Computadores

Curso 2012/2013

Penas Sabín, Darío <dario.penas@udc.es>
Pereira Guerra, Adrián <adrian.pereira@udc.es>

## Índice

# 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:

```
// Uso del bucket en radix sort
while (m / exp > 0){
  int bucket[10] = {0};
  for (i = 0; i < n; i++){
    bucket[a[i] / exp %10]++;
  }
  for (i = 1; i < 10; i++)
    bucket[i] += bucket[i - 1];
  for (i = n - 1; i >= 0; i--)
    b[--bucket[a[i] / exp %10]] = a[i];
  for (i = 0; i < n; i++){
    a[i] = b[i];
  }
  exp *= 10;
}
```

## 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: 1$\underline{2}$ 9$\underline{2}$
3: 3$\underline{3}$
4:
5: 2$\underline{5}$
6: 8$\underline{6}$
7: 5$\underline{7}$ 3$\underline{7}$
8: 4$\underline{8}$
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 el 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, pero es necesario que todos los procesos conozcan este dato, ya que influye en el número de ejecuciones del algortmo. Esto se hace con MPI_Allreduce, que realiza la operación indicada (MPI_MAX en este caso) 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);
```

También es necesario compartir el bucket en cada iteración, puesto que se recalcula cada vez, y se ordena en función de su suma acumulada.

El trabajo de realizar la suma acumulativa lo hace el proceso 1 y le manda el resultado al 0.

Al utilizar el bucket acumulado para calcular las posiciones ordenadas de los elementos del array final, se hace desde el final al principio del array a ordenar por lo que, al paralelizarlo, es necesario secuencializar 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 $n-2$, ...).

```
1  // Usando el bucket se guardan en b los elementos de a ordenados
2  for (i = n - 1; i >= 0; i--){
3
4    int pos = (--bucket[a[i] / exp % 10]);
5
6    rank_recv = floor(pos / n);
7    rank_send = myrank;
8
9    if( rank_send == rank_recv ){
10     if(myrank==1)
11       pos = pos %n;
12     b[ pos ] = a[i];
13   }else{
14     if(myrank==0)
15       pos = pos %n;
16     a_intercambio [pos] = a[i];
17   }
18 }
```

# 3. Paralelización con OpenMP

# 4. Conclusiones

## 4.1. Opinión personal

# A. Código secuencial

```c
#include <math.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define MAX 100000000

void radixsor(int *a, int n){
int i, m = a[0], exp = 1;
int *b = malloc(MAX*sizeof(int)+1);


for (i = 0; i < n; i++){
  if (a[i] > m){
    m = a[i];
  }
}
while (m / exp > 0){
  int bucket[10] = {0};
  for (i = 0; i < n; i++){
    bucket[a[i] / exp % 10]++;
  }
  for (i = 1; i < 10; i++)
    bucket[i] += bucket[i - 1];
  for (i = n - 1; i >= 0; i--)
    b[--bucket[a[i] / exp % 10]] = a[i];
  for (i = 0; i < n; i++){
    a[i] = b[i];
  }
  exp *= 10;
  }

}


int main(int argc, char* argv[]){
  int numElementos = atoi(argv[1]);
  int i;
  int *array = malloc(MAX*sizeof(int)+1);
  static int n;

  for (n = 0; n < numElementos; n++){
    int r = rand()%1000000;
    array[n] = r;
  }
  radixsor (&array[0], numElementos);

  return 0;
}
```

./src/secuencial.c

## B. main con MPI

```
int main(int argc, char* argv[]){
  int numElementos = atoi(argv[1]);
  int i, p, myrank;
  MPI_Status status;
  struct timeval t0, t1, t;

  if(argc < 2){
    printf("Usage: mpirun -n numprocs ./radix numElementos \n");
    exit(1);
  }
  /* Inicializacion de MPI*/
  MPI_Init (&argc, &argv);
  /* myrank will contain the rank of the process
  MPI_COMM_WORLD is all processors together */
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  /* p es el numero de procesos */
  MPI_Comm_size (MPI_COMM_WORLD, &p);
  /* n es el numero de elementos de cada proceso*/
  int n = numElementos/p;
  int *a = malloc(n*sizeof(int));
  inicializar_a_creciente(&a[0], n, myrank);
  /* Ejecuta el algoritmo de ordenamiento*/
  radixsort_parallel (a, numElementos, p, myrank);

  if (myrank == 0)
      MPI_Recv(a, n, MPI_INT, 1, 123, MPI_COMM_WORLD, &status);
    else
    MPI_Send(a, n, MPI_INT, 0, 123, MPI_COMM_WORLD);

  free(a);
  MPI_Finalize ();
}
```

./src/main.c

## C. Implementación radixsort con MPI

```
void radixsort_parallel(int* a, int elemTot, int p, int myrank){
  int i, m, exp = 1;
  struct timeval t0, t1, t;
  int rank_send, rank_recv;
  int n = elemTot/p;
  int pos_envio = -1;
  int* b = malloc(sizeof(int)*n);
  int* a_intercambio = malloc(sizeof(int)*elemTot);
  int* bucket = malloc(sizeof(int)*10);
  int otro_rank = (myrank+1) %p;
  int iter = 0;
  MPI_Status status;

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

```
72
73        for(i=0;i<n;i++){
74          if(a_intercambio[i]!=-1){
75            b[i] = a_intercambio[i];
76          }
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