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>
https://github.com/adrisons/AEC

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: 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 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

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, &
         status);
4    MPI_Recv(&bucket[0], 10, MPI_INT, 1, 4, MPI_COMM_WORLD, &status);
5    for(i=0;i<elemTot;i++){
6      if(a_intercambio[i]!=-1){
7        // Se adapta la posicion del otro proceso
8        pos_envio = i %n;
9        b[pos_envio] = a_intercambio[i];
10     }
11   }
12   inicializar_array(&a_intercambio[0], elemTot, -1);
13 }
```

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 quedaran esperando mútuamente 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`.

Como se puede ver en la gráfica de la Figura 1, no se consiguen mejoras en este ejemplo usando `MPI` con dos procesadores. Esto se debe al gran paso de datos entre procesadores, la secuencialización de un trozo del código y el hecho de sólo usar dos procesadores. Además, cuando mayor sea el máximo elemento del array a ordenar, más iteraciones tiene que hacer el algoritmo, mayor paso de datos en la paralelización `MPI` y mayor ineficiencia.

numElementos	t_seq(seg)	t_mpi(seg)	speedup_mpi
10	0.0000	1.1847	0.00
500	0.0001	1.0277	0.00
1000	0.0002	1.0910	0.00
5000	0.0016	1.0918	0.00
10000	0.0021	1.0310	0.00
50000	0.0108	1.0499	0.01
200000	0.0422	1.1182	0.04
500000	0.1043	1.2588	0.08
1000000	0.2122	1.4812	0.14
5000000	1.0474	3.3106	0.32
10000000	2.0751	5.5255	0.38
50000000	10.2957	23.5374	0.44
100000000	20.6674	46.3649	0.45

./res/tabla_mpi

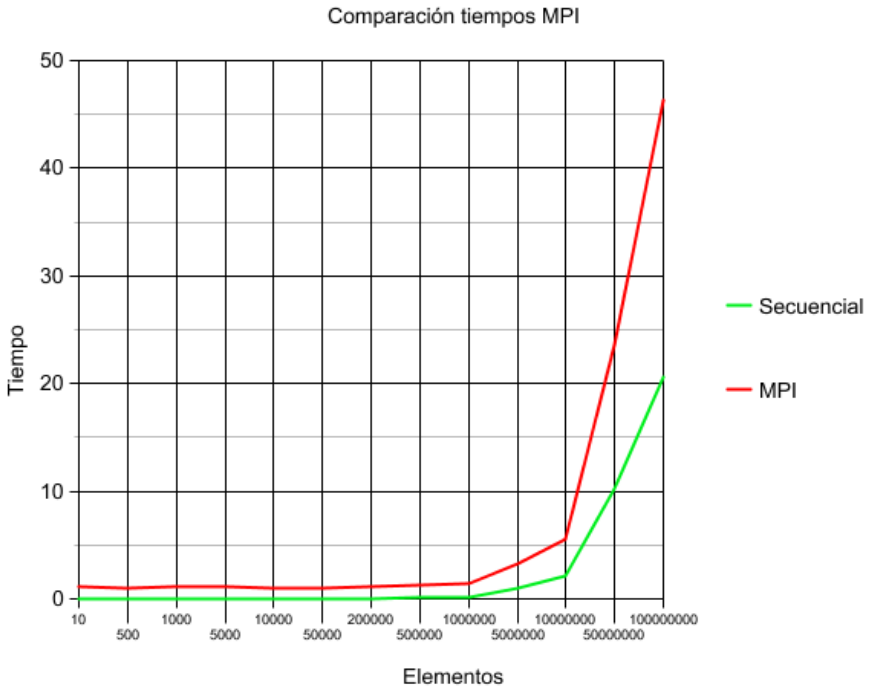

Figura 1: Comparación ejecuciones secuencial y `MPI`

3. Paralelización con OpenMP

Como se comprueba en las pruebas, al emplear `MPI` sobre este problema no se obtiene ningún beneficio. Por ello, se aplicará `OpenMP` sobre el algoritmo secuencial en vez de hacerlo sobre el código paralelizado con `MPI`.

Como se había aclarado en el apartado de `MPI`, hay partes de nuestro programa que no son paralelizables (o con una complejidad demasiado elevada para el objetivo de este curso). A continuación se muestran partes del código sobre las que sí se puede aplicar `OpenMP`.

Al inicializar el array, por ejemplo, podemos dividir el trabajo entre varios threads.

```
1  #pragma omp parallel
2  {
3    #pragma omp for schedule(static) private(i)
4    for (n = 0; n < numElementos; n++){
5      int r = rand()%100;
6      array[n] = r;
7    }
8  }
```

./src/inicializacion_openmp.c

Al hallar el máximo elemento del array también se nota mejoría al utilizar varios hilos.

```
1  #pragma omp parallel
2  {
3    #pragma omp for schedule(static) private(i)
4    for (i = 0; i < n; i++){
5      if (a[i] > m){
6        m = a[i];
7      }
8    }
9  }
```

./src/maximo_openmp.c

Por último, al copiar los datos del array auxiliar b en el array principal a podemos usar varios hilos, ya que los datos no se pisan.

```
1    #pragma omp parallel
2    {
3      #pragma omp for schedule(static) private(i)
4      for (i = 0; i < n; i++){
5        a[i] = b[i];
6      }
7    }
```

./src/copia_datos.c

Aplicando estas pequeñas modificaciones se aprecia una gran mejora en el tiempo de ejecución con respecto al original (Figura 2 y Figura 3), y eso que la parte principal no se puede paralelizar (la que hace uso del bucket).

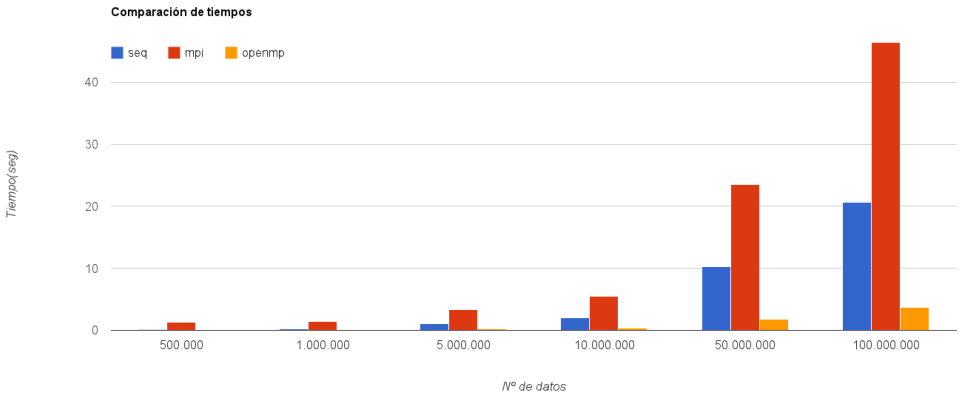

Figura 2: Comparación tiempos

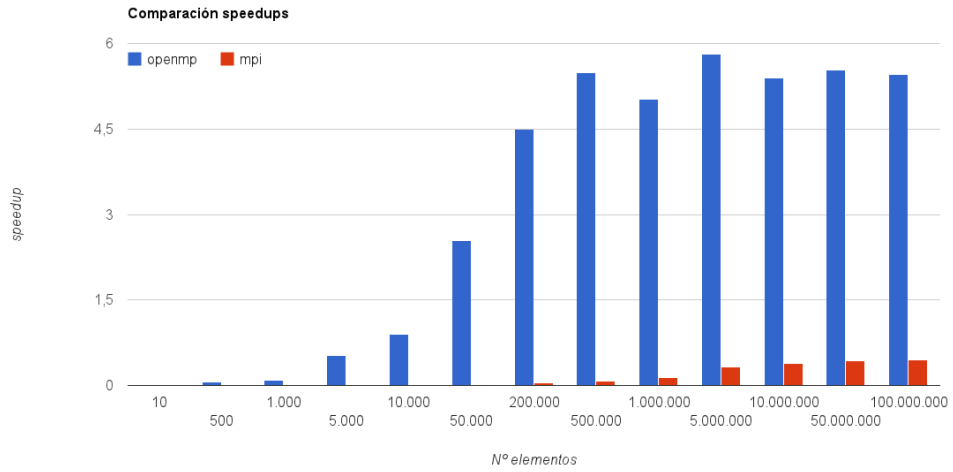

Figura 3: Comparación speedups

4. Conclusiones

Esta práctica nos ha resultado bastante complicada, ya que el problema era bastante complicado y no nos dimos cuenta de todas las restricciones en un principio. De todos modos, nos ha parecido muy interesante ver que se pueden mejorar mucho los tiempos de ejecución de ciertos programas, aunque pueda costar bastante esfuerzo.

Nuestra percepción general de la asignatura es positiva, debido sobre todo a la segunda parte que, en vez de ver el paralelismo desde un punto de vista teórico, experimentamos a trabajar realmente con él y aprendemos más, en nuestra opinión. Preferimos aprender las cosas de este modo que no viéndolo sólamente de manera teórica en clase.

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

```c
int main(int argc, char* argv[]){
  int numElementos = atoi(argv[1]);
  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,
            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 P1
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 P0 envia el array de intercambio al P1 para continuar
            ordenando
64      if(myrank==0){
65        MPI_Send(a_intercambio, elemTot, MPI_INT, 1, 5,
              MPI_COMM_WORLD);
66      }
67      // El proceso 1 envia los elementos que le corresponden del
            proceso 0
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

D. main con OpenMP

```
1  int main(int argc, char* argv[]){
2    int numElementos = atoi(argv[1]);
3    int i;
4    int *array = malloc(MAX*sizeof(int)+1);
5    static int n;
6    struct timeval t0, t1, t;
7
8
9    assert (gettimeofday (&t0, NULL) == 0);
10   #pragma omp parallel
11   {
12     #pragma omp for schedule(static) private(i)
13     for (n = 0; n < numElementos; n++){
14       int r = rand()%100;
15       array[n] = r;
16     }
17   }
18   radixsor (&array[0], numElementos);
19
20   assert (gettimeofday (&t1, NULL) == 0);
21   timersub(&t1, &t0, &t);
22   printf("%d.%06ld", (long int)t.tv_sec, (long int)t.tv_usec);
23
24   return 0;
25 }
```

./src/main_openmp.c

E. Implementación radixsort con OpenMP

```
1  void radixsor(int *a, int n){
2    int i, m = a[0], exp = 1;
3    int *b = malloc(MAX*sizeof(int)+1);
4
5    #pragma omp parallel
6    {
```

```
7       #pragma omp for schedule(static) private(i)
8       for (i = 0; i < n; i++){
9         if (a[i] > m){
10          m = a[i];
11        }
12      }
13    }
14    while (m / exp > 0){
15      int bucket[10] = {0};
16
17      for (i = 0; i < n; i++){
18        bucket[a[i] / exp %10]++;
19      }
20
21      for (i = 1; i < 10; i++)
22        bucket[i] += bucket[i - 1];
23
24      for (i = n - 1; i >= 0; i--)
25        b[--bucket[a[i] / exp %10]] = a[i];
26
27      #pragma omp parallel
28      {
29        #pragma omp for schedule(static) private(i)
30        for (i = 0; i < n; i++){
31          a[i] = b[i];
32        }
33      }
34      exp *= 10;
35    }
36
37  }
```

./src/radixsort_openmp.c