Capítulo 3

Técnica de Particionado

En esta tesis se separa el concepto de técnica de particionado de lo que es el algoritmo de ordenación en sí. Esto lo hacemos así porque aplicamos la misma técnica de particionado tanto para el algoritmo de ordenación secuencial como para el paralelo. Una vez realizado el particionado, los datos se ordenan con un algoritmo de ordenación secuencial.

Por un lado, aplicando el particionado se busca reducir los fallos en la jerarquía de memoria en el algoritmo secuencial (esté o no incluido en un algoritmo paralelo). Por otro lado, se busca equilibrar la carga de trabajo y la comunicación para el algoritmo paralelo.

En este Capítulo primero se exponen, en detalle, los objetivos que se quieren alcanzar con las técnicas de particionado propuesta en esta tesis. A continuación se hace un repaso de otras técnicas de particionado que utilizan distintas formas de particionar (de repartir los datos en conjuntos disjuntos). El particionado de los datos está íntimamente ligado con el algoritmo de ordenación. Así, en algunos casos, explicamos la técnica de particionado implícita en algunos métodos de ordenación. Después se proponen dos técnicas de particionado, Reverse Sorting y Counting Split, que publicamos en su momento como parte de algoritmos específicos para claves de 32 y 64 bits, respectivamente. Finalmente, se ve que estas técnicas se pueden usar tanto para claves de 32 y 64 bits. Esto se consigue combinándolas en una única técnica conjunta, Mutant Reverse Sorting, que proponemos en esta tesis, y que no distingue entre claves de diferente número de bits y, además, es consciente de la distribución de datos que se tienen que ordenar. Mutant Reverse Sorting es consciente de las ventajas y desventajas de Reverse Sorting y de Counting Split. Esta técnica consiste en realizar un particionado de Reverse Sorting sobre un muestreo de datos y, en función del particionado que se consiga del muestreo, decide particionar todo el conjunto con Reverse Sorting, o por lo contrario decide particionar con Counting Split (muta a Counting Split).

3.1. Objetivos

En el Capítulo 2 se vieron los inconvenientes de Radix sort en la ordenación secuencial y paralela. En el caso de la ordenación secuencial, la ordenación con Radix sort puede tener un elevado número de fallos en la jerarquía de memoria cuando el conjunto de datos a ordenar es relativamente grande. En el caso de la ordenación paralela, además de los inconvenientes de la ordenación secuencial, el volumen de comunicación de datos puede ser muy grande.

Una solución a estos problemas es particionar el conjunto de datos a ordenar en subconjuntos cuyos datos estén ordenados entre sí (los datos de cada subconjunto pertenecen a un determinado rango de valores disjunto y se conoce el orden relativo entre subconjuntos). Después del particionado, cada subconjunto se ordena independientemente.

En realidad, con el particionado se busca obtener un algoritmo de ordenación que sea eficiente para cualquier tipo de datos. Para ello, se deben conseguir los siguientes objetivos en la ordenación:

1. **Explotar la localidad temporal de los datos.**

   Cuanto más pequeño es el conjunto de datos a ordenar, más probabibilidad hay de que quepa en un nivel de memoria cache más cercano al procesador. Así, aunque el conjunto de datos a ordenar aumente, el número de fallos de acceso a memoria por elemento no aumentará excesivamente, es decir, se estará explotando la jerarquía de memoria del procesador. Esta explotación de la jerarquía de memoria se puede conseguir tanto cuando se está ordenando en secuencial como cuando se está ordenando en paralelo, y hace que el rendimiento del algoritmo no se degrade con conjuntos grandes.

2. **Ahorrar parte de la ordenación de la clave de los datos a ordenar.**

   Los datos pertenecientes a un subconjunto de datos están en un determinado rango de valores y, por consiguiente, todas las claves tienen parte de los bits más significativos iguales. Así, al ordenar ese subconjunto de datos con el algoritmo Radix sort (que es el caso de esta tesis) este prefijo común no tiene que ordenarse. Con datos con mucho sesgo o datos con muchos duplicados esto puede significar un ahorro importante de ordenación tanto a nivel secuencial como paralelo. Conociendo el número de bits comunes y ordenando con el algoritmo Radix sort, ahorramos trabajo.

3. **Reducir la cantidad de datos comunicados.**

   En el algoritmo paralelo, como los subconjuntos de datos están ordenados entre sí, los procesadores sólo necesitan comunicarse entre ellos para obtener las

particiones. Después, sólo es preciso una ordenación local de los datos de cada partición.

4. **Equilibrar la carga de trabajo.**

   El particionado debe ser de tal forma que la distribución de los datos entre los procesadores sea lo más equilibrada posible.

Para hacer una partición eficiente de los datos se tiene que tener en cuenta el tipo de datos (32 ó 64 bits), el tamaño del conjunto de datos que se van a ordenar, su distribución, y además, qué estructuras adicionales se van a necesitar para poder hacer dicha partición.

En esta tesis, se propone una técnica de particionado eficiente, independientemente del tipo de distribución de datos, con tal de obtener un algoritmo de ordenación basado en el algoritmo de Radix sort que cumpla con los objetivos planteados. Para que sea eficiente, el coste de hacer la partición del conjunto en subconjuntos más el coste de ordenar cada subconjunto, no debe superar el coste de ordenar el conjunto de datos directamente con algún otro método de ordenación.

3.2. Trabajo relacionado con técnicas de particionado

En esta sección se explican tres algoritmos de ordenación, Bucket sort, MSB-Radix sort y Sample sort, que particionan el conjunto de datos en subconjuntos de datos ordenados entre sí; aunque ninguno de los tres métodos soluciona ni busca los mismos o todos los objetivos de esta tesis.

No se realizará un análisis en profundidad de estos algoritmos, entre otras cosas porque no son los métodos de ordenación paralela más rápidos. Por ejemplo, el algoritmo Load Balanced Radix sort, presentado en [40], es hasta un 30 % mejor que la versión paralela de Bucket sort y Sample sort (MSB-Radix sort no tiene versión paralela). Más adelante, en el Capítulo 5, se compararán las propuestas de esta tesis con el trabajo presentado en [40]. Sin embargo, se quieren explicar estos tres algoritmos porque lo propuesto en esta tesis se basa en una combinación de ideas de estos métodos. Bucket sort y Sample sort son anteriores a este trabajo, MSB-Radix sort es contemporáneo a nuestro trabajo.

Se explicará la versión secuencial de los algoritmos por motivos de claridad.

Bucket sort

En el NAS benchmark para ordenación de enteros se utiliza una versión paralela de Bucket sort [6]. El objetivo final de este Bucket sort no es obtener el vector ordenado

sino obtener la posición (orden) de cada una de las claves en el vector, si éste estuviera ordenado por la clave. Así, el objetivo es obtener un vector con posiciones relativas r_0, r_1 a r_{n-1} para las claves k_0, k_1 a k_{n-1}, de tal forma que si $r_i < r_j$ entonces $k_i < k_j$; donde n es el número total de claves a ordenar y $0 \leq j, i < n$. No obstante, la adaptación a ordenación es evidente.

El Algoritmo 3 muestra, desde un punto de vista secuencial, el funcionamiento básico de Bucket sort [30]. Éste es un algoritmo que sigue los siguientes pasos:

1. **Paso (1) - Inicialización de estructuras:** Inicializa la estructura auxiliar D donde se van a guardar temporalmente las particiones. Esta estructura D puede ser, por ejemplo:
   - una lista encadenada de datos de clave y puntero para cada partición.
   - una matriz de nb x n datos de clave y puntero. Las particiones se guardan por filas y cada una de las nb particiones debería poder almacenar n datos de clave y puntero, puesto que inicialmente no se sabe la distribución de los datos y es posible que todos los datos vayan a una única partición.
2. **Paso (2) - Particionado:** Para cada clave de b bits del vector S a ordenar, se obtiene el valor que representan sus bits de mayor peso b_{m-1} con la función *clasifica*. Una vez obtenido el valor de esos bits, la clave se almacena adecuadamente con la función *guarda*, junto al resto de claves con el mismo valor para esos bits, en la estructura D. Esto se realiza en las líneas 4 a 7. El número de particiones depende del número de bits b_{m-1} tal y como sucede en el algoritmo de Radix sort. En este caso, el número de particiones es de $nb = 2^{b_{m-1}}$.
3. **Paso (3) - Ordenación de las particiones:** Se ordena cada partición almacenada en la estructura D. Cada partición se puede ordenar con cualquier método de ordenación y de forma independiente ya que al ordenar por los bits de mayor peso, las claves de dos particiones diferentes ya están ordenadas entre si. Líneas 9 a 11.

MSB-Radix sort

MSB-Radix sort [34, 30] es un algoritmo secuencial (no tiene versión paralela) que es contemporáneo a nuestro trabajo. Este trabajo tiene similitudes con la técnica de particionado Reverse Sorting, que proponemos en esta tesis. Las similitudes y diferencias se comentarán, en detalle, en la Sección de Reverse Sorting.

MSB-Radix sort [34, 30], al igual que Bucket sort, particiona el conjunto de datos basándose en los bits de mayor peso de la clave a ordenar. El Algoritmo 4 muestra el

Algoritmo 3: Algoritmo de Bucket sort(S, n, b)

1: – **Paso 1:** Inicialización de la estructura auxiliar D.
2: $inicializa(D, nb, n)$

3: – **Paso 2:** Particionado
4: **para** $i = 1$ a n **hacer**
5: $value \leftarrow clasifica(S[i], b, b_{m-1})$
6: $guarda(D, S[i], value)$
7: **fin para**

8: – **Paso 3:** Ordenación de las particiones.
9: **para** $ib = 1$ a nb **hacer**
10: $ordena(D, ib)$
11: **fin para**

código para MSB-Radix sort para ordenar n elementos de clave y puntero de b bits cada uno.

El algoritmo de MSB-Radix distingue dos casos:

- **Caso (1) - Ordenación de conjuntos pequeños:** Si no hay que ordenar nada de la clave ($b <= 0$) entonces se acaba (líneas 2 a 4 del Algoritmo 4). Si el conjunto de datos es suficientemente pequeño, éste se ordena con el algoritmo de ordenación de *Straight Insertion* [30], líneas 5 a 6. El algoritmo de *Straight Insertion* es un algoritmo de ordenación basado en comparaciones que es muy eficiente para conjuntos de datos muy pequeños.

- **Caso (2) - Particionado de conjuntos no pequeños:** Se ordena el conjunto de datos por el dígito de mayor peso con tal de particionar estos datos (Paso (2.1)) y se aplica MSB-Radix sort a cada una de las particiones (Paso (2.2)). Esto se realiza en las líneas 10 a 19 del Algoritmo 4.

  - **Paso (2.1) - Particionado:** El particionado se hace con el algoritmo de conteo (que se ha explicado en el Capítulo 2 para Radix sort), llamando a la función *particiona_conteo* para los b_{m-1} bits de mayor peso de los b bits. D es un vector auxiliar de datos de clave y puntero donde se guardan las particiones. C guarda el número de claves y punteros para cada partición.

    Con MSB-Radix sort, a diferencia de la ordenación con Radix sort, el algoritmo de conteo se aplica a los bits de mayor peso b_{m-1} de los b bits de la clave que faltan por ordenar. Además, el número de bits de mayor peso utilizados puede variar, a diferencia de Bucket sort donde b_{m-1} es siempre

de un tamaño fijo. Para MSB-Radix sort, el número de bits de mayor peso b_{m-1} por los que se ordenan las claves es igual a $\text{mín}(\log_2(n) - 2, 16)$. Así, el tamaño de este dígito varía en función del número de claves a ordenar, y está limitado por 16 bits, un número de bits calculado empíricamente por los autores de [34].

- **Paso (2.2) - Aplicar MSB-Radix sort a cada partición:** En este momento se aplica otra vez MSB-Radix sort a cada una de estas particiones almacenadas en el vector D, líneas 13 a 18. Las claves pertenecientes a una misma partición están consecutivas en el vector D.

Algoritmo 4: Algoritmo de MSB_Radix sort (S, D, n, b)

1: – **Caso (1):** Ordenación conjuntos pequeños.
2: **si** $b \leq 0$ **entonces**
3: Nada
4: **fin si**

5: **si** $n \leq STRAIGHT_INSERTION_BREAK$ **entonces**
6: $Straight_Insertion(n)$
7: **sino**
8: – **Caso (2):** Particionado de conjuntos no pequeños.
9: – **Paso (2.1):** Particionado.
10: $b_{m-1} = \text{mín}(\log_2(n) - 2, 16)$
11: $< D, C > \leftarrow particiona_conteo(S, n, b, b_{m-1})$

12: – **Paso (2.2)** Aplicar MSB-Radix sort a cada partición.
13: $off \leftarrow 0$
14: $nbuckets \leftarrow 2^{b_{m-1}}$
15: **para** $ib = 1$ a nb **hacer**
16: $MSB_Radix(D + off, S + off, C[ib], b - b_{m-1})$
17: $off \leftarrow off + C[ib]$
18: **fin para**
19: **fin si**

MSB-Radix sort es un algoritmo recursivo que puede hacer más de un paso de particionado del conjunto de datos; a diferencia de Bucket sort, que sólo realiza un paso de particionado. De esta forma, MSB-Radix sort quiere evitar tener que ordenar particiones grandes. Bucket sort sólo realiza una vez el particionado, y por consiguiente, es posible que tenga que ordenar particiones grandes.

Sample sort

Sample sort es un algoritmo que propone una forma de particionar los datos basada en el muestreo de los datos a ordenar. En este caso, la filosofía es totalmente diferente a la de Bucket sort y MSB-Radix sort. Hay muchos trabajos distintos sobre este tipo de ordenación, por ejemplo [13, 15], todos ellos algoritmos paralelos. Sin embargo, por claridad, se explicará una implementación secuencial. El Algoritmo 5 sigue los siguientes pasos:

1. **Paso (1) - Muestreo:** Se cogen, aleatoriamente, q claves de los elementos a ordenar del vector fuente y se almacenan en un vector que se llamará *vector_muestreo*. La distribución de estas q claves representan la distribución de los elementos a ordenar. Esto se hace en las líneas 2 a 4 del Algoritmo 5.

2. **Paso (2) - Ordenación del vector de muestreo:** Se ordenan las p claves del muestreo realizado. Se puede utilizar cualquier método de ordenación. Esto se hace en la línea 6 del Algoritmo.

3. **Paso (3) - Creación del vector de separadores:** Se crea un vector de separadores con $s - 1$ claves equidistantes en el vector de muestreo ya ordenado. Líneas 8 a 11 del Algoritmo 5. Como el vector de muestreo es representativo del conjunto a ordenar, se puede decir que estas $s - 1$ claves también dividirán en particiones más o menos iguales el vector original cuando esté ordenado. Cada separador es el primer elemento de una partición.

4. **Paso (4) - Particionado:** Para cada clave de los elementos a ordenar se realiza una búsqueda dicotómica en el vector de separadores. Con ello, se quiere averiguar a qué partición pertenecen las claves. Una vez se sabe a qué partición pertenece la clave, se guardan clave y puntero en la estructura auxiliar D. Todo este proceso se realiza en las líneas 13 a 16 del Algoritmo.

5. **Paso (5) - Ordenación:** Finalmente se pueden ordenar cada una de las particiones guardadas en la estructura auxiliar D con algún método de ordenación (líneas 18 a 20 del Algoritmo).

La versión paralela de este algoritmo incorpora varios pasos de comunicación para obtener el vector de separadores y varios pasos para comunicar los datos. Hay autores que proponen otras formas de escoger las claves de separadores, como la llamada *Exact Splitting*[13]. Estas formas de obtener los separadores sólo tienen sentido en la versión paralela. *Exact Splitting* consiste en ordenar primero los conjuntos de datos locales que hay en cada procesador, después se escogen unos primeros candidatos a claves separadoras del vector local ordenado y, finalmente, entre todas esas claves candidatas y todos los procesadores, se eligen finalmente las claves separadoras para todos los

Algoritmo 5: Algoritmo de Sample sort(S, n)

1: – **Paso 1:** Muestreo.
2: **para** $i = 1$ a q **hacer**
3: $\quad vector_muestreo[i] \leftarrow S[random()].clave$
4: **fin para**

5: – **Paso 2:** Ordenación muestreo.
6: $vector_muestreo \leftarrow Ordenacion(vector_muestreo, q)$

7: – **Paso 3:** Creación vector separadores.
8: $desp \leftarrow q/2s$
9: **para** $i = 1$ a $s - 1$ **hacer**
10: $\quad vector_separadores[i] \leftarrow vector_muestreo[desp + (i-1)q/s]$
11: **fin para**

12: – **Paso 4:** Particionado.
13: **para** $i = 1$ a n **hacer**
14: $\quad value \leftarrow busqueda_binaria(S[i], vector_separadores)$
15: $\quad guarda(D, S[i], value)$
16: **fin para**

17: – **Paso 5:** Ordenación.
18: **para** $ib = 1$ a nb **hacer**
19: $\quad S \leftarrow Ordenacion_particion(D, ib)$
20: **fin para**

procesadores. Con esas claves separadoras se obtienen particiones locales que después se comunican a los procesadores que le correspondan.

Análisis de los algoritmos explicados

MSB-Radix sort y Bucket sort obtienen un particionado equitativo si el rango de valores para los dígitos de mayor peso sigue una distribución uniforme. En caso contrario, que los datos queden mejor o peor repartidos depende de cuan sesgados sean los valores que tomen las claves para los dígitos de mayor peso. Cuanto más reducido sea el rango de valores que pueden tomar las claves y mayor es el número de datos del conjunto, más desequilibrio entre las particiones habrá. En el caso de Bucket sort este desequilibrio puede tener más importancia ya que solamente se realiza un particionado del conjunto de claves y punteros a ordenar. Si hay desequilibrio, el

algoritmo ordenará particiones de tamaños muy diferentes. El algoritmo MSB-Radix sort, sin embargo, continúa con la ordenación por los dígitos de mayor peso hasta que encuentra conjuntos de datos muy pequeños, por lo que soluciona el desequilibrio entre particiones. No obstante, esto puede significar un coste elevado para conjuntos grandes de claves y punteros, ya que puede tener que hacer muchos particionados.

Además, MSB-Radix sort utiliza el algoritmo de conteo sin tener en cuenta el tamaño del dígito por el que particionará. Este dígito, de hecho, es mayor cuanto mayor es el conjunto a ordenar. Se debe recordar que una de las causas del mal rendimiento de Radix sort son los fallos de TLB cuando el conjunto de datos es grande y el número de bits del dígito b_i, que se está ordenando, cumple que 2^{b_i} es mayor que el número de entradas del TLB.

Sample Sort es el método que ofrece mejores garantías para conseguir una distribución equitativa de los datos entre los subconjuntos. Eligiendo adecuadamente los parámetros q y s (ver [13]), el algoritmo asegura que, con una probabilidad determinada que se puede elegir, se obtendrá una distribución equitativa de los datos. Sin embargo, en conjuntos con duplicados y con un número reducido de valores diferentes en estos duplicados, la probabilidad de tener duplicados en el vector de muestreo y en los separadores extraídos de ese muestreo es elevada. En consecuencia, al tener duplicados en los separadores, cuando se reparten los datos según los rangos que delimiten los separadores, habrá particiones que podrán tener un número de datos mayor que el ideal, $\frac{n}{s}$. El número de particiones s que se van a formar se escoge según el del número de procesadores y la intención de obtener un equilibrio de carga entre procesadores.

Por lo tanto, el rendimiento de cada uno de estos algoritmos depende del tamaño y de la distribución de datos que se esté ordenando. Por consiguiente, se puede decir que **ninguno** de estos algoritmos ofrece, **a la vez**, una **buena solución** a los objetivos planteados. De esta manera, es posible que no se explote la jerarquía de memoria cuando se ordena alguna de las particiones.

En esta tesis, se proponen dos técnicas, Reverse Sorting y Counting Split, que solucionan en parte los inconvenientes que se observan en los algoritmos Bucket sort, MSB-Radix sort, y Sample sort. Además, a la vista de los inconvenientes que también se pueden encontrar con estas dos técnicas, se propone una técnica que combina las ventajas de Reverse Sorting y de Counting Split que llamamos Mutant Reverse Sorting.

A continuación se empezará a ver Reverse Sorting, después Counting Split y finalmente la técnica que las combina. Para estas técnicas se explican la versión secuencial y paralela.

3.3. Reverse Sorting

En esta sección se explica la técnica de particionado Reverse Sorting que publicamos en [28] y analizamos en distintos contextos en [23, 25] como parte de un algo-

ritmo secuencial y paralelo. Estos algoritmos secuenciales y paralelos se presentarán en los Capítulos 4 y 5.

3.3.1. Algoritmo Secuencial

El pseudocódigo del algoritmo de particionado secuencial se muestra en el Algoritmo 6.

Los pasos que sigue esta técnica de particionado son los siguientes:

- **Paso (1) - Particionado:** Para particionar el conjunto de datos se ordena el vector de datos de clave y puntero por el dígito de mayor peso de la clave (dígito $m-1$), utilizando el algoritmo de conteo explicado en el Capítulo 2. Este particionado se realiza en el pseudocódigo del Algoritmo 6 con la función *particiona_conteo* tal y como se explica en la sección de MSB-Radix sort. El número de particiones obtenidas depende del número de bits del dígito $m-1$. Si el número de bits del dígito es b_{m-1}, el número de particiones que se obtendrá es $2^{b_{m-1}}$, que son tantas particiones como los posibles valores de ese dígito. Así, las claves pertenecientes a una partición, tienen los b_{m-1} bits de mayor peso todos iguales. Si se quiere obtener el vector ordenado faltará ordenar el resto de bits $b-b_{m-1}$ de las claves de cada partición, independientemente. Por ejemplo, esto se podría ordenar con Radix sort.

- **Paso (2) - Particionado recursivo de algunas particiones:** Para cada partición obtenida, se mira si el número de elementos es mayor que R_{TLB} o R_{cache_i}. R_{TLB} es la cantidad de memoria que puede mapear el TLB, expresada en número de datos de clave y puntero. R_{cache_i} es la capacidad de memoria del nivel i de cache, expresada también en número de datos de clave y puntero. Si se supera R_{TLB} o R_{cache_i}, y faltan todavía bits para ordenar de la clave, se aplica otra vez Reverse Sorting a esa partición (Línea 8 del Algoritmo). En caso contrario, no se particionará otra vez. La idea aquí es que, posteriormente, al ordenar con Radix sort, no se tengan los problemas que se plantearon en el Capítulo 2 con la jerarquía de memoria ya que las particiones que se ordenarán con Radix sort cabrán en el nivel de cache y no se tendrán problemas de conflictos de TLB.

Como se comentó al comienzo del Capítulo, Reverse Sorting y MSB-Radix sort (contemporáneo a nuestro trabajo) tienen similitudes, pero hay dos puntos importantes que distinguen Reverse Sorting de MSB-Radix sort:

1. **El momento en el que se deja de particionar.**

   Reverse Sorting deja de particionar cuando el tamaño del conjunto de datos es menor que R_{cache_i} y R_{TLB}, a diferencia de MSB-Radix sort que deja de particionar

Algoritmo 6: Algoritmo de Reverse Sorting (S, D, n, b)

1: – **Paso (1)** Particionado del vector S.
2: $< D, C > \leftarrow particiona_conteo(S, n, b, b_{m-1})$

3: – **Paso (2)** Particionado Recursivo de ciertas particiones.
4: $off \leftarrow 0$
5: $nbuckets \leftarrow 2^{b_{m-1}}$
6: **para** $ib = 0$ a $nbuckets - 1$ **hacer**
7: **si** $(C[ib] > R_{TLB} \vee C[ib] > R_{cache_i}) \wedge (b - b_{m-1} > 0)$ **entonces**
8: $Reverse_Sorting(D + off, S + off, C[ib], b - b_{m-1})$
9: **fin si**
10: $off \leftarrow off + C[ib]$
11: **fin para**

cuando decide ordenar con *Straight Insertion*, es decir, cuando las particiones tiene menos de 32 datos (línea 5 del Algoritmo 4).

El objetivo de Reverse Sorting es hacer que las estructuras de datos que se necesitan para ordenar una partición con Radix sort quepan en la memoria cache y que pueda ser mapeable por la estructura del TLB. Con ello, se quieren evitar los problemas de Radix sort con la jerarquía de memoria, planteados en el Capítulo 2. Si el conjunto es muy pequeño, el algoritmo de ordenación que usa Reverse Sorting, también ordenaría con *Straight Insertion*, como MSB-Radix sort.

2. **Elección del tamaño del dígito del Reverse Sorting.**

El número de bits que se elige para los dígitos de Reverse Sorting es b_r (antes llamado b_{m-1}) y depende del número de entradas del TLB del computador y del tamaño del primer nivel de cache. Los autores de Bucket Sort paralelo [1] ya tuvieron en cuenta el número de entradas del TLB a la hora de particionar los datos. Aquí, además, la elección del número de bits también tiene en cuenta los estudios realizados para Radix sort, en el Apéndice A. Según estos estudios, la memoria necesaria para los contadores no debería superar el tamaño del primer nivel de cache.

En particular, en esta tesis, se define b_r tal que cumpla que $b_r < \log_2 E_{TLB} 2^{b_r} * T_{cnt} < C_1$. E_{TLB} es el número de entradas del TLB del procesador, T_{cnt} es el tamaño en bytes de un contador y C_1 es la capacidad en bytes del primer nivel de cache. Así pues, b_r es constante para un determinado computador, a diferencia que sucede en MSB-Radix sort, donde depende del número de elementos a ordenar.

Con todo esto, el objetivo de Reverse Sorting es conseguir:

a) Minimizar el número de fallos de primer nivel de memoria cache por el acceso a los contadores.

b) Minimizar el número de fallos accediendo a la estructura TLB cuando se realiza el particionado.

Como se explicó para el algoritmo de conteo (Capítulo 2), si el número de páginas activas de memoria cuando se escribe en el vector D es más grande que el número de entradas de TLB, se tendrá un número alto de fallos accediendo al TLB.

Por otra parte, el tener un b_r fijo para cada computador, podría provocar que se tuviera que realizar un número significativo de pasos de particionado para conjuntos grandes de datos o claves con valores sesgados.

- Para conjuntos grandes, esto puede suceder si $\log_2 E_{TLB}$ es muy pequeño, que no es el caso de los procesadores actuales.
- Para conjuntos con claves con valores sesgados, aplicar repetidamente Reverse Sorting a las particiones puede ser ineficiente. Este problema se soluciona más adelante cuando se proponga la técnica combinada de Reverse Sorting y Counting Split.

Como se explicará en el Capítulo 4 de análisis de algoritmos secuenciales de ordenación, el método que se propone ofrece mejor rendimiento que MSB-Radix sort para conjuntos con una distribución relativamente uniforme de los datos. Para conjuntos donde el sesgo de los datos es notable, Reverse Sorting puede tener peor rendimiento. Este caso se soluciona más adelante.

Evaluación del Algoritmo

En las Figuras 3.1 y 3.2 se muestran los CSE de ejecuciones reales para el particionado de datos de clave y puntero de 32 y 64 bits respectivamente para distribuciones de datos Random (R), S20 y D100 para el procesador Power4 en el computador basado en p630 (líneas continuas en las gráficas).

Los CSE según el modelo desarrollado en esta tesis, se muestran con líneas discontinuas en las gráficas. Para las distribuciones S20 y D100, los CSE según el modelo se calculan a partir del número de particionados que se deben realizar en una ejecución real. En la gráfica de la derecha de ambas Figuras, se muestran los CSE acumulados para todas las llamadas recursivas (Paso (1)+Paso(2) del Reverse Sorting), para los pasos del algoritmo de conteo (ver Sección 2.1 para más detalle) utilizado en Reverse Sorting. Así, *Mov* es el paso de movimiento, *S. Par.* es el paso de suma parcial de contadores, *Count* es el paso de Conteo e *Ini*, el paso de inicialización de contadores.

Los CSE son para la ordenación de conjuntos de datos para los que las estructuras ya no caben en el segundo nivel de cache del Power4. Como se puede observar, los CSE de las ejecuciones y del modelo tienen tendencias parecidas:

- Para claves de 32 bits (Figura 3.1), los CSE para Random y S20 son los mismos hasta $2M$ datos. Para el Power4, $br = 9$, por lo que para S20, sólo es preciso una iteración de Reverse Sorting para conseguir particionar los datos de forma equitativa. A partir de $2M$ datos, para la distribución S20 se tienen que hacer 2 iteraciones de Reverse Sorting. Sin embargo, para D100, son necesarias 4 iteraciones de Reverse Sorting, por lo que el número de CSE es mayor para los conjuntos de datos ordenados.

  Por otra parte, se observa un pequeño aumento del número de CSE para D100 y Random en los resultados experimentales cuando el número de elementos es aproximadamente $2M$ datos. Con S20 no se distingue ya que para ese conjunto de datos también se incrementa el número de iteraciones de Reverse Sorting necesarias. Este aumento de CSE para 2M datos para D100 y Random se debe a que en ese momento las estructuras necesarias en la ordenación no caben en el tercer nivel de cache. Por consiguiente, las dos lecturas del vector fuente, una para realizar el paso de conteo y otra para el paso de movimiento, podrían tener que buscar los datos a memoria. La penalización por fallo de memoria cache de estas dos lecturas no es muy grande gracias al prefetch hardware de que dispone el procesador Power4.

- Para claves de 64 bits (Figura 3.2), el número de CSE para S20 y D100 es mayor que para la distribución Random. El número de iteraciones de Reverse Sorting necesarias para S20 y D100, para poder particionar adecuadamente los datos, son 2 y 7 respectivamente.

  Al igual que sucede con claves de 32 bits, también hay un aumento de CSE debido a que las estructuras necesarias no caben en el tercer nivel de memoria cache para conjuntos de datos superiores a $1M$ datos.

Se puede observar que el modelo (que desarrollamos en detalle en el Apéndice C) se aproxima bastante al comportamiento del particionado con Reverse Sorting, en función de la distribución de datos a ordenar.

En la Tabla 3.1 se muestran los fallos de primer nivel de cache y TLB para una distribución Random. Además, se muestra el número de accesos al segundo y tercer nivel de memoria cache y a memoria principal para el paso de conteo y de movimiento. Para 32 bits, se muestran los accesos y fallos para ordenar 4M datos. Para 64 bits, se muestran resultados para 2M datos. En la tabla no se muestran los resultados para S20 y D100 porque son muy similares. Como era de esperar, el número de fallos y accesos es casi el doble para 64 bits que para 32 bits.

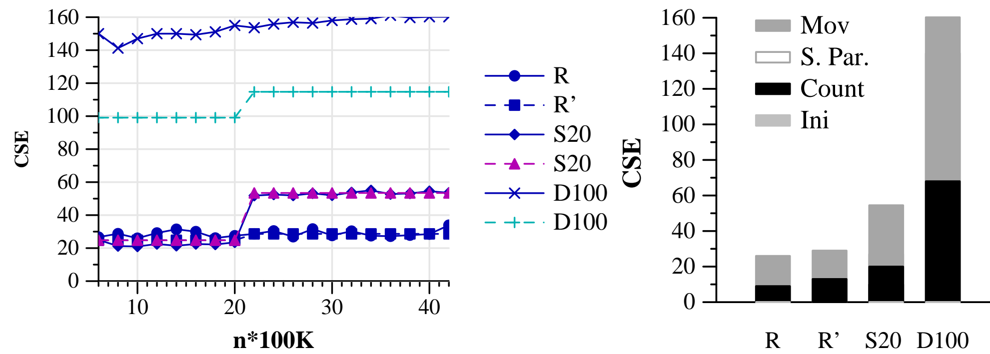

Figura 3.1: CSE experimentales en un Power4 (continua) y del modelo (discontinua) para Reverse Sorting para distribuciones Random, S20 y D100, de 100K a 4M datos de 32 bits (izquierda). CSE para 4M datos divididos en los CSE de los diferentes pasos del algoritmo de conteo utilizado en el particionado (derecha).

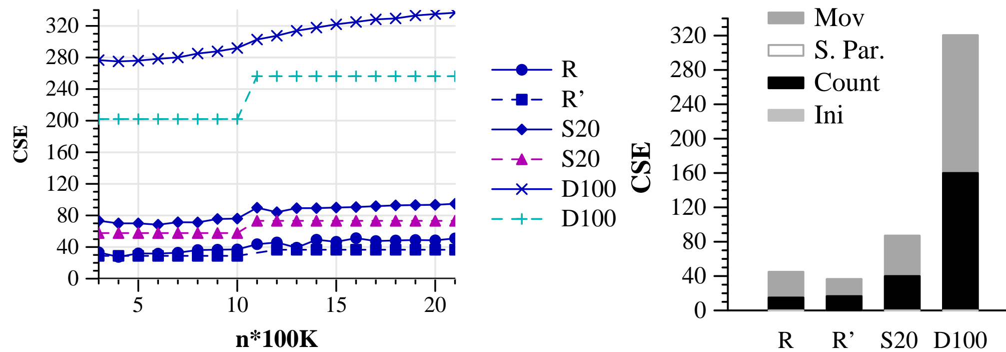

Figura 3.2: CSE experimentales en un Power4 (continua) y del modelo (discontinua) para Reverse Sorting para distribuciones Random, S20 y D100, de 100K a 2M datos de 64 bits (izquierda). CSE para 2M datos divididos en los CSE de los diferentes pasos del algoritmo de conteo utilizado en el particionado (derecha).

	Power4			
	32 bits		64 bits	
Fallos y Accesos /n	*C*	*M*	*C*	*M*
F. Primer nivel	0,013	0,033	0,028	0,065
A. Segundo nivel	0,059	0,070	0,122	0,156
A. Tercer nivel	0,002	0,002	0,003	0,004
A. Mem. Principal	0,009	0,008	0,017	0,015
Modelo Mem. P.	0,008	0,008	0,016	0,016
F. TLB	0,002	0,018	0,004	0,026
Modelo TLB	0,002	0,002	0,004	0,004

Tabla 3.1: Fallos de acceso al primer nivel de cache y de traducción del TLB, y número de accesos al segundo y tercer nivel de memoria cache y memoria principal dividido entre el total de elementos a ordenar, 4M datos(32bits) y 2M datos (64bits) para una distribución Random. C y M, en la tabla, son el paso de conteo y de movimiento respectivamente.

En la misma tabla, se muestran los cálculos para los accesos a memoria y fallos de TLB, a partir del modelo (Apéndice C). Éstos son los dos aspectos que pueden influir más significativamente en el aumento del número de CSE.

Los cálculos según el modelo, para los accesos de memoria y fallos de TLB en el paso de conteo son bastante ajustados a los obtenidos en las ejecuciones. Sin embargo, para los fallos de TLB del paso de movimiento, los cálculos según el modelo, son inferiores a los resultados obtenidos. Esto es debido a que el modelo no ha contabilizado el número de fallos por conflicto. En cualquier caso, el número de fallos de TLB es bajo y no influye significativamente en el número de CSE.

Por lo tanto, se puede concluir que:

1. Si la distribución de los datos a particionar es más o menos uniforme, el número total de CSE de Reverse Sorting no será muy elevado ya que el número de iteraciones a realizar será pequeño. Sin embargo, para distribuciones con mucho sesgo (D100 o S20) el rendimiento de Reverse Sorting empeora ya que el número de iteraciones es mayor que para las distribuciones Random.

2. Durante el particionado con Reverse Sorting, se está explotando la jerarquía de memoria sin incurrir en un número elevado de fallos de TLB ni de memoria cache.

3.3.2. Algoritmo Paralelo

El algoritmo de particionado paralelo con Reverse Sorting (Algoritmo 7) sigue los siguientes pasos:

1. **Paso (1) - Reverse Sorting local:** Cada procesador aplica una iteración de Reverse Sorting a sus datos locales. Para ello, se utiliza el dígito de mayor peso (de b_r bits) de los b bits que todavía quedan por ordenar de las claves locales. Esto se hace en la línea 3 del Algoritmo.

2. **Paso (2) - Comunicación de los contadores:** Los procesadores se intercambian los 2^{b_r} contadores locales obtenidos durante Reverse Sorting local. De esta forma, los procesadores pueden saber el número total de datos de clave y puntero que, en el conjunto de los P procesadores, hay para cada partición global. Cada procesador P_{pid} contribuirá con una porción de sus datos locales B_i^{pid} para conseguir en el Paso (5) una partición global B_i, tal y como se muestra en la Figura 3.3, $B_i = \sum_{pid=0}^{P-1} B_i^{pid}$ para $0 \leq i < 2_r^b$ (línea 5 del Algoritmo).

3. **Paso (3) - Cálculo de distribución de particiones globales:** Con los contadores recibidos y los contadores locales, cada procesador calcula la asignación de particiones globales a procesadores. Esto se realiza en la línea 7 del Algoritmo 7. Si no se consigue un buen equilibrio de carga entre los procesadores se vuelven a realizar los pasos (1) y (2) por los siguientes bits de mayor peso de la claves ($b_r = b_{m-2}$). Se dice que se ha conseguido equilibrio de carga entre los procesadores si el número de claves K para cualquier grupo de particiones consecutivas asignadas a un procesador es $K < n/p + U$. U es el desequilibrio de carga permitido, el cual es de un mínimo de 100K datos por procesador y un máximo de $\text{máx}(100K, n/p^2)$.

   El número de particiones globales que le correspondan a cada procesador puede variar con tal de obtener un buen equilibrio de carga. Línea 7 del Algoritmo 7.

4. **Paso(4) - Renombramiento de procesadores lógicos a procesadores reales:** Para reducir la cantidad de comunicación, todos los procesadores deciden un renombramiento de sus identificadores lógicos en el proceso de la ordenación. Con ello se quiere aprovechar la mayor localidad posible de los grupos de particiones asignados a los procesadores. Así, a aquel procesador que tenga mayor número de datos pertenecientes al grupo de particiones i, que debe ser asignado a un procesador con identificador lógico i, se le asignará el identificador lógico i. Si el procesador ya tiene asignado otro identificador lógico, se buscará el siguiente procesador con mayor número de datos pertenecientes a ese grupo de particiones. Esto se efectúa en la línea 10 del Algoritmo 7.

5. **Paso (5) - Comunicación de datos:** Las particiones locales son enviadas a los procesadores a los que se ha asignado estas particiones en el Paso (4). Este paso de comunicación se realiza con una comunicación *All to All* (línea 12 del Algoritmo).

Algoritmo 7: Reverse Sorting Paralelo

1: **repetir**
2:   – **Paso (1):** Reverse Sorting local
3:   $< D, C > \leftarrow particiona_conteo(S, n, b, b_r)$

4:   – **Paso (2):** Comunicación de los contadores
5:   $GlC \leftarrow allgather(C, 2^{b_r})$

6:   – **Paso (3):** Cálculo de distribución de particiones
7:   $equilibrio_carga \leftarrow distribucion_particiones(Gl)$
8: **hasta** $equilibrio_carga <> CIERTO$

9: **Paso (4):** Renombramiento de Procesadores
10: $renombre \leftarrow renombramiento(Gl)$

11: **Paso (5):** Comunicación de datos
12: $comunicacion_datos(S, D, Gl, renombre)$

La elección de b_r se realiza en función del número de entradas de TLB de los procesadores. Si este número es menor que el número de procesadores con los que se quiere hacer la ordenación se puede actuar de dos formas:

1. Realizar dos o más iteraciones de Reverse Sorting con los datos locales a cada procesador.

2. Realizar una iteración de Reverse Sorting con un dígito, con un número de bits mayor que b_r, con el coste de penalización de fallos de TLB que eso puede implicar.

Aquí se ha optado por realizar dos o más iteraciones si es necesario.

La reasignación de identificadores de procesadores lógicos antes de realizar la comunicación de los datos es importante si hay sesgo local de los datos. Se dice que hay sesgo local cuando hay particiones locales a un procesador que tienen más elementos que otras. Que haya sesgo local no significa que haya sesgo en el global de los datos. A continuación se muestra cómo el sesgo local puede influir en la cantidad total de datos comunicados y cómo se puede reducir dinámicamente.

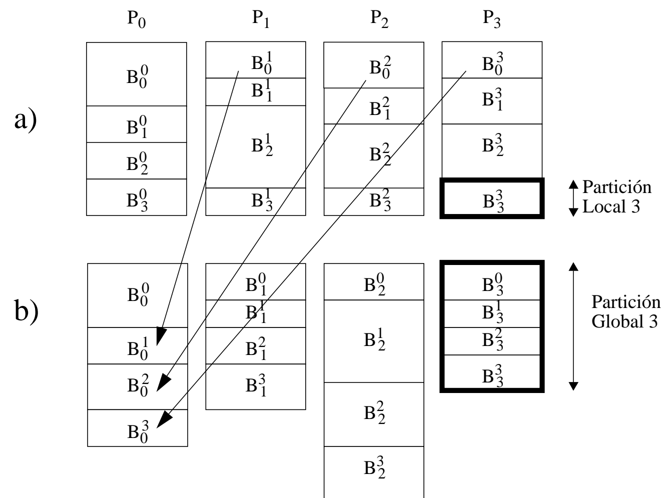

Figura 3.3: Formación de particiones globales a partir de las particiones locales de cada procesador; para 4 particiones y 4 procesadores.

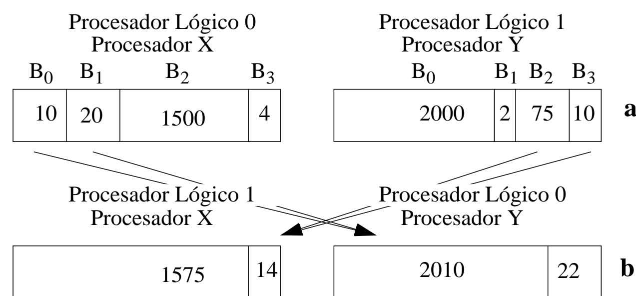

Figura 3.4: Ahorro de comunicación entre procesos.

La Figura 3.4.a muestra un ejemplo para dos procesadores X e Y a los que se llaman Procesador lógico[1] 0 y 1 respectivamente. Dicha figura ilustra un ejemplo de Reverse Sorting de una sola iteración con $b_{m-1} = 2$. Se va a suponer que al final de una iteración de Reverse Sorting el Procesador lógico 0 tiene que almacenar en su memoria el conjunto de claves con valores más bajos (particiones B_0 y B_1), y que el Procesador Lógico 1 tiene que almacenar el conjunto de claves con valores más altos (particiones B_2 y B_3). Aquí también se supone que si un procesador tiene que almacenar más de una partición, éstas almacenan claves con los valores consecutivos en los b_{m-1} bits de mayor peso, como en la implementación que se realiza en estas tesis.

En la Figura 3.4.a, se puede observar que el Procesador lógico 0 tiene 10, 20, 1500 y 4 claves en cada una de las 4 particiones formadas durante Reverse Sorting local. Las particiones 0 y 1 quedan asignadas al Procesador lógico 0 para el resto de la ordenación, mientras que las particiones 2 y 3 quedan asignadas al Procesador lógico 1. Reasignando el Procesador lógico 0 al Procesador Y y el Procesador lógico 1 al Procesador X sólo hay que comunicar 85 claves del Procesador Y al Procesador X, y 30 claves en la dirección inversa, tal y como se puede observar en la Figura 3.4.b. Si no se reasignaran los Procesadores Lógicos, se acabarían comunicando 1504 y 2002 claves entre procesadores.

Esta estrategia tan simple implica una pequeña cantidad de trabajo y no acarrea ninguna consecuencia, en el sentido de que los datos quedan finalmente ordenados si se mantiene un vector de orden para determinar qué identificador lógico tiene cada Procesador real. Por lo tanto, esta reasignación de identificadores lógicos puede ahorrar una gran cantidad de comunicación de datos a través de la red.

Evaluación del Algoritmo

En la Figura 3.5 se muestran los CSE del particionado cuando se realizan 1, 2 ó 3 iteraciones de Reverse Sorting. En esta figura se distinguen los CSE (acumulados) para los pasos más significativos del particionado paralelo. Los CSE son para el particionado de 4M datos (arriba) y 4M datos por procesador (abajo) de clave y puntero de 64 bits para 2 a 16 procesadores. La división de los CSE que se muestra es, de arriba a abajo: CSE de preparación de los mensajes (prepare), CSE de comunicación de datos correspondientes al Paso (5) (All to All), CSE para la comunicación de los contadores del Paso (2) (Transpose), CSE para el cálculo y evaluación de la repartición de particiones en el Paso (3) del algoritmo (Eval) y CSE para Reverse Sorting local, Paso (1) del algoritmo (Reverse Local). No se muestran los resultados para 32 bits porque las conclusiones que se obtienen son las mismas. En las gráficas se muestran, con líneas discontinuas, los CSE según el modelo que se explica en el Apéndice C. El

[1]En MPI, a cada proceso se le asigna una numeración entre 0 y $P-1$ con tal de que los procesos se puedan identificar entre ellos. Esta numeración no tiene nada que ver con el procesador físico.

modelo muestra la misma tendencia que los resultados obtenidos.

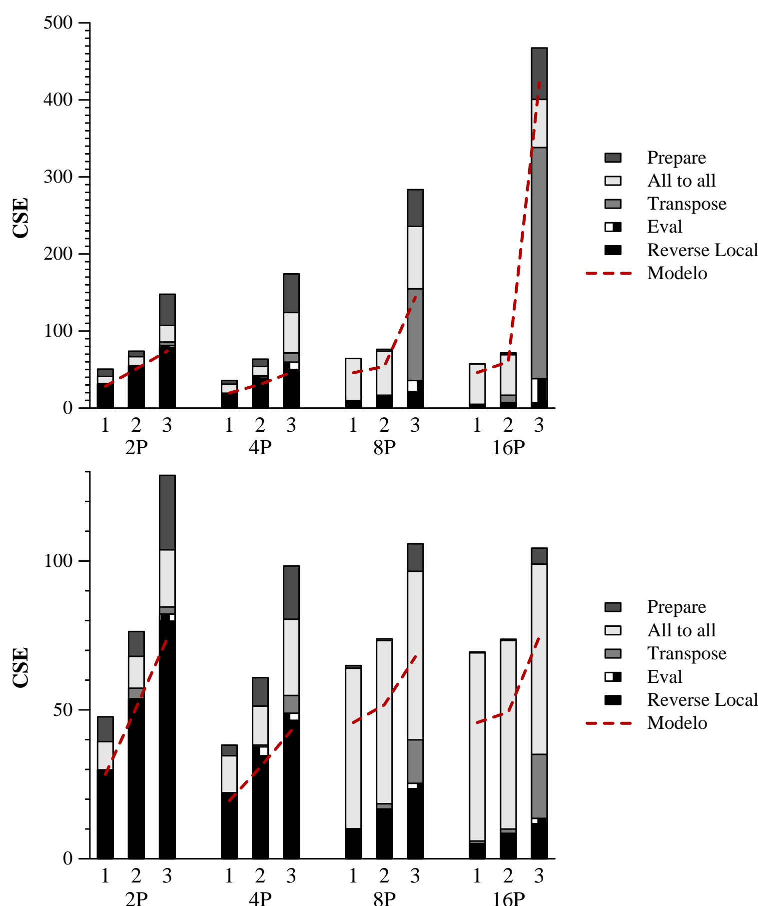

Figura 3.5: CSE invertidos en los pasos de Reverse Sorting paralelo para 4M datos (arriba) y 4PM (4M por procesador, abajo) datos de clave y puntero de 64 bits con el computador basado en módulos p630 usando de 2 a 16 procesadores. Se muestran los resultados para cuando Reverse Sorting realiza 1,2, ó 3 iteraciones.

Se empezará comentando los CSE para el particionado de 4M datos de clave y puntero. Para un mismo número de procesadores, a medida que se aumenta el número de iteraciones de Reverse Sorting a realizar, los CSE de comunicación de contadores y Reverse Sorting local, los pasos (1) y (2) del particionado respectivamente, aumentan. Los CSE de comunicar los contadores crecen de forma más significativa que los CSE de Reverse Sorting local. Esto es debido a que el número de contadores a comunicar crece exponencialmente con el número de iteraciones que se realizan. En ambos casos, el aumento del número total de CSE se hace significativo a partir de la tercera iteración de Reverse Sorting.

Por otro lado, para un mismo número de iteraciones de Reverse Sorting, los CSE de comunicación de contadores aumentan a medida que aumenta el número de procesadores. Esto se hace más visible cuando se hacen tres iteraciones en el computador basado en p630. El particionado con Reverse Sorting obtiene 512 particiones en la primera iteración, y en cada iteración adicional el número total de particiones se multiplica por 32. El número total de contadores crece de la misma forma. Por lo tanto, en la tercera iteración el número total de contadores a comunicar es de 512K contadores. De ahí que el número de CSE sea significativo.

Sin embargo, los CSE debido a Reverse Sorting local se decrementan a medida que aumenta el número de procesadores ya que el número de datos por procesador es menor, pudiéndose explotar mejor la jerarquía de memoria.

En cualquier caso, lo que se puede observar es que el computador basado en p630 no permite mucha escalabilidad ya que en cuanto el número de procesadores es mayor que 4, la comunicación entre estos procesadores se ralentiza (pasa de $1500MB/s$, entre procesadores del mismo nodo, a $350MB/s$ entre procesadores de diferentes nodos). Es por ello que hay un aumento de CSE en el paso *AlltoAll* cuando se pasa de 4 a 8 procesadores.

Para la gráfica de abajo de la Figura 3.5 se muestra cuál es el comportamiento si el número de elementos a ordenar es 4M datos por procesador. Los CSE debido a Reverse Sorting local se reducen a medida que el número de procesadores aumenta, tal y como sucede para ordenar 4M entre todos los procesadores. El número de ciclos invertidos por procesador en su Reverse Sorting local es el mismo ya que hay $\frac{n}{P} = 4M$ datos por procesador, pero el número total de elementos ordenados entre todos los procesadores es $\frac{n}{P}P = n$, por lo que el número de CSE relativo es 2 veces menor con respecto al anterior número de procesadores en la gráfica.

En cuanto a los CSE de comunicar los contadores, el aumento que se produce al incrementar el número de procesadores es sensiblemente menor. Esto es porque el tamaño del problema aumenta con el número de procesadores y el peso de comunicar más y más contadores no es tan significativo. Sin embargo, a partir de la cuarta iteración, el número de contadores a comunicar es tan elevado que vuelve a ser significativo.

En el modelo no se tuvieron en cuenta dos aspectos que se muestran en la Figura. Estos no influyen significativamente en la tendencia de los CSE. Son aspectos que dependen de la implementación que se tenga de la librería MPI.

1. El primero de estos aspectos es el número de CSE de preparación de la recepción del mensaje (*prepare* en las Figuras). La preparación de la estructura (tipo de MPI) para recibir el mensaje ayuda a que los datos enviados desde cada procesador se guarden directamente en la posición final del vector destino, de tal forma que datos pertenecientes a una misma partición global se guardan consecutivos[2]. Cuantas más iteraciones se hacen de Reverse Sorting, más particiones

[2]Quizás, otro tipo de implementación, no tan estándar, hubiera ayudado a mejorar este aspecto.

y más significativo se hace el número de CSE de preparación de tipos. En esta preparación, la cantidad de trabajo y accesos a memoria aumentan significativamente de 1 a 3 iteraciones de Reverse Sorting. En cambio, estos CSE son menos significativos cuantos más datos de clave y puntero se tienen que ordenar. Así, cuando el número de elementos a ordenar por procesador es fijo, los CSE de preparación disminuyen a medida que aumentamos el número de procesadores, como se puede observar en la gráfica de abajo de la Figura 3.5.

2. El segundo aspecto que no se ha modelado es el número de CSE debido a la forma de recibir los datos. Al recibir los datos, éstos se copian directamente en la partición que corresponda, utilizando el tipo definido con MPI. Así, el número de CSE de comunicación de un mismo número de datos, aumenta con el número de iteraciones para un mismo procesador. Cuanto mayor número de iteraciones, mayor número de particiones, y por consiguiente, las estructuras del tipo de datos definido para almacenar las claves y punteros se hacen más y más grandes. Además, es probable que se esté escribiendo en un número elevado de posiciones de memoria distintas, una por cada partición que se está creando. Esto último puede ser causa de un número elevado de fallos de TLB.

Finalmente, queremos ver el equilibrio de carga obtenido para dos distribuciones de datos: Random y Gaussian. No estudiamos qué sucede con el resto de distribuciones de datos analizadas en este documento porque se sabe que serían necesarias más de 2 iteraciones y por consiguiente el número de CSE sería significativamente alto. En la Figuras 3.6 y 3.7 se muestra el tanto por ciento de desequilibrio de carga obtenido para diferentes números de procesadores (de 2 a 16 procesadores) cuando se ordena un conjunto de datos para una distribución Random y Gaussian respectivamente. Aquí, el desequilibrio de carga se ha calculado como la diferencia media del número total de elementos de cada procesador con el número ideal de elementos n/P al final del particionado. En las gráficas de la izquierda de las Figuras se muestra el desequilibrio de carga cuando el número total de datos a ordenar es de 4M y en las gráficas de la derecha se muestra el desequilibrio de carga cuando *cada procesador* tiene que ordenar 4M de datos. Como se puede observar, el desequilibrio de carga obtenido es pequeño aunque no perfecto. El equilibrio de carga, para estas distribuciones, se obtuvo después de una iteración de Reverse Sorting.

Para concluir, el particionado paralelo con Reverse Sorting ofrece un muy buen rendimiento siempre y cuando no se realicen más de dos iteraciones de Reverse Sorting. Además, el equilibrio de carga para grandes conjuntos de datos está garantizado, lo cual es uno de los objetivos de esta tesis.

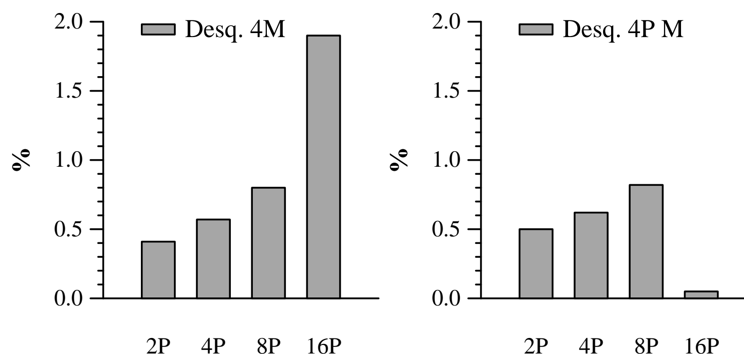

Figura 3.6: Desequilibrio de carga según la desviación estándar para 4M datos a ordenar (izquierda) y $4P$M datos a ordenar (derecha). Resultados para una distribución Random en el computador basado en p630, variando el número de procesadores.

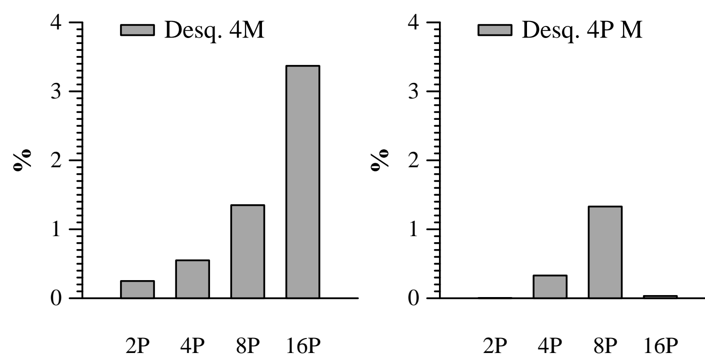

Figura 3.7: Desequilibrio de carga según la desviación estándar para 4M datos a ordenar (izquierda) y $4P$M datos a ordenar (derecha). Resultados para una distribución Gaussian en el computador basado en p630, variando el número de procesadores.

3.4. Counting Split

El nombre de Counting Split viene de la idea de combinar el vector de separadores (*split vector* en inglés) propuesto para Sample sort [13] y el algoritmo de conteo [30] (*Counting Algorithm* en inglés). Con Counting Split se proponen mejoras al algoritmo de particionado Sample sort, que usa muestreos y que se presentó en la introducción de este Capítulo (Sample sort).

3.4.1. Algoritmo Secuencial

El Algoritmo 8 muestra el algoritmo secuencial de Counting Split que consiste básicamente en los siguientes pasos:

1. **Paso (1) - Obtención del vector de muestreos:** Se obtiene, de forma aleatoria, q claves del vector a ordenar S y se forma el vector de muestreo (línea 2 del Algoritmo).

2. **Paso (2) - Ordenación del vector de muestreos:** Se ordena el vector de muestreo con Radix sort (línea 4 del Algoritmo).

3. **Paso (3) - Creación del vector de separadores:** Se crea el vector de separadores con $s-1$ claves. Para ello, se elige $s-1$ claves equidistantes del vector de muestreo ordenado, empezando en la clave q/s-ésima del vector de muestreo (Línea 6 del Algoritmo).

4. **Paso (4) - Particionado con el algoritmo de conteo:** Se usa el algoritmo de conteo para particionar el vector de datos usando cada una de las $s-1$ claves del vector de separadores como delimitadores de las particiones que se van a crear. Más abajo se explica en qué consiste aplicar el algoritmo de conteo con el vector de separadores. Aquí, además, con estas $s-1$ claves separadoras se forman $2s$ particiones ya que para el algoritmo propuesto, dos claves consecutivas en el vector de separadores delimitan dos particiones. Primero, se determina que una clave k está entre dos separadores $vector_separadores[i]$ y $vector_separadores[i+1]$ si cumple que:

   $$vector_separadores[i] \leq k < vector_separdores[i+1]$$

   Después, se distinguirá entre si esta clave pertenece a una partición impar o par según el siguiente criterio:

   - **Partición impar:** partición de claves iguales al separador i, es decir:

     $$k = vector_separadores[i].$$

- **Partición par:** partición de las claves mayores que el separador i, es decir $k > vector_separadores[i]$, pero menor que $vector_separdores[i + 1]$.

El algoritmo de conteo, utilizando el vector de separadores, aparece en el Algoritmo 8 como los Pasos (4,1), (4,2), (4,3) y (4,4). Estos pasos consisten en:

a) **Paso (4.1) - Inicialización de Contadores:** Inicialización de los $2s$ contadores (línea 8 del Algoritmo 8).

b) **Paso (4.2) - Conteo:** Para cada clave del vector a ordenar se realiza una búsqueda dicotómica en el *vector_separadores*. Será en este momento cuando se determine si la clave pertenece a una partición par o impar. Además, a parte de incrementar el contador de la partición a la que se asignará el dato, el número de partición a la que será asignado se guarda en el vector *indice_particion*. Guardando este índice se evita tener que hacer una segunda búsqueda dicotómica cuando se realice el paso de movimiento Paso (4.4). El Paso (4.2) se realiza en las líneas 10 a 14 del algoritmo.

c) **Paso (4.3) - Suma Parcial:** Se realiza la suma parcial de los $2s$ contadores (líneas 16 a 22 del Algoritmo).

d) **Paso (4.4) - Movimiento:** Se leen, otra vez, las claves del vector a ordenar y se copian en el vector destino, en la partición correspondiente, que se conoce gracias al vector *indice_particion* (líneas 24 a 28 del Algoritmo).

Nótese que el coste de la búsqueda dicotómica puede ser elevado. Para reducir dicho coste se ha hecho una implementación que intenta aprovechar al máximo las optimizaciones que puede hacer el compilador. Para ello:

- Se expresa la búsqueda dicotómica con condiciones **si/sino**. Es decir, si se tiene que decidir a que partición va una clave, se tendrán las sentencias condicionales **si/sino** anidadas necesarias hasta decidir que la clave va a una determinada partición. Esto reduce el número total de saltos que realizaría una implementación de una búsqueda binaria con un bucle **mientras**. Además, hay procesadores que tienen instrucciones de movimientos condicionales, como el R10K, que puede sacar provecho de este tipo de códigos, con instrucciones si/sino.

- Cada uno de los separadores del *vector_separadores* se guarda en una variable. Con esto, si hay suficientes registros del procesador para guardar todos los separadores, se ahorra tener que hacer accesos a memoria accediendo al *vector_separadores*, cada vez que se quieran realizar comparaciones en la búsqueda dicotómica con el vector de separadores.

En la sección de evaluación del algoritmo, más adelante, se compara esta implementación de búsqueda dicotómica con la implementación tradicional con un bucle mientras.

Algoritmo 8: Algoritmo de Counting Split(S, n, q, s)

1: – **Paso (1):** Vector de muestreos
2: $vector_muestreo \leftarrow muestreo_aleatorio(S, q)$

3: – **Paso (2):** Ordenación de muestreos
4: $Radix_sort(vector_muestreo, q)$

5: – **Paso (3):** Creación vector separadores
6: $vector_separadores \leftarrow obtener_separadores(vector_muestreo, q, s)$

7: – **Paso (4.1)** Inicializa Contadores
8: $inicializa(C, 2s)$

9: – **Paso (4.2):** Conteo
10: **para** $i = 0$ a $n - 1$ **hacer**
11: $value \leftarrow busqueda_binaria(S[i], vector_separadores)$
12: $indice_particion[i] \leftarrow value$
13: $C[value] \leftarrow C[value] + 1$
14: **fin para**

15: – **Paso (4.3):** Suma Parcial
16: $tmp \leftarrow C[0]$
17: $C[0] \leftarrow 0$
18: **para** $i = 0$ a $2s - 2$ **hacer**
19: $accum \leftarrow tmp + C[i]$
20: $tmp \leftarrow C[i + 1]$
21: $C[i + 1] \leftarrow accum$
22: **fin para**

23: – **Paso (4.4):** Movimiento
24: **para** $i = 0$ a $n - 1$ **hacer**
25: $value \leftarrow indice_particion[i]$
26: $D[C[value]] \leftarrow S[i]$
27: $C[value] \leftarrow C[value] + 1$
28: **fin para**

Una vez se han particionado los datos, habrá una parte de la clave perteneciente a las particiones $2i$ (pares) que no hará falta que se ordene. A diferencia de Reverse Sorting, el número de bits ya ordenados para las claves pertenecientes a una partición puede diferir del número de bits ya ordenados en las claves de otra parti-

ción. Para saber cuántos bits hay ya ordenados en una partición únicamente se tiene que calcular cuantos bits comunes hay entre los separadores *vector_separadores*$[i]$ y *vector_separadores*$[i+1]$. Las claves pertenecientes a las particiones $(2i+1)$(impares) ya están ordenadas; al ser todas ellas iguales, tienen todos los bits iguales.

En cualquier caso, se mantiene la propiedad de estabilidad de los datos, es decir, las claves duplicadas también mantienen el orden relativo. Otros autores proponen añadir *tags* [10] a de las claves a ordenar, que podrían ser los punteros asociados a las claves que se tienen aquí, con tal de tenerlos en cuenta, de alguna forma, en la ordenación. Con el método propuesto en esta tesis, no se tiene que ordenar más que lo que realmente se quiere ordenar, las claves de los datos de clave y puntero.

Parámetros q y s

El número de particiones y de muestreos que se quieren realizar, s y q respectivamente, son dos parámetros importantes en Counting Split. El criterio para la elección de s y q es diferente al tomado en las propuestas de Sample sort. Los criterios, aquí, son los siguientes:

1. s está limitado por E_{TLB} (número de entradas de TLB) al igual que sucede con el número de particiones realizadas con Reverse Sorting. Sin embargo, aquí el coste de particionado es proporcional a $O(n \log(s))$, por lo que también se deberá ser consciente de ello a la hora de decidir el número de particiones con E_{TLB} grandes (como es el caso del computador basado en p630). Aunque E_{TLB} es $2^{10} = 1024$, aquí se toma un número más reducido.

2. q se calcula con la ayuda de la Fórmula 3.1 propuesta en [13].

$$q = s * \frac{2\ln(s/p)}{(1-1/\beta)^2\beta} \tag{3.1}$$

   donde p es la probabilidad de que cualquiera de las s particiones tenga más de $\beta n/s$ claves. β es el factor de desequilibrio permitido ($\beta > 1$). Sin embargo, los valores de q obtenidos directamente de esta fórmula pueden ser muy grandes dependiendo del número máximo de claves que se quiere en una ordenación. Por ejemplo, si se quiere conseguir una buena distribución de los datos, con una probabilidad del 88 % ($p = 0{,}12$), en 32 particiones con un desequilibrio entre particiones de como máximo el 4 % ($\beta = 1{,}04$), se precisaría realizar un total de $200K$ muestreos. Dependiendo del número de datos de clave y puntero a ordenar, este puede ser un coste significativo.

   Aquí se va a fijar el número de muestreos en $q = 4096$, con lo que con una probabilidad alta del 75 % se obtendrá un desequilibrio menor del 25 % con respecto a los $\frac{n}{s}$ datos que le debería corresponder a cada partición, según la fórmula.

Evaluación del Algoritmo

Primero se quiere justificar porqué se hizo una implementación totalmente desenrollada de la búsqueda dicotómica para el Counting Split. En la Figura 3.8 se muestran los CSE para la implementación de la búsqueda dicotómica con condiciones si/sino explícita y una implementación con bucle, cuando se particiona un conjunto de 2M datos de clave y punteros de 64 bits y una distribución Random. Como ya se comentó en la sección anterior, esta implementación reduce el número de saltos totales y, además, permite la optimización del código por parte del compilador: los separadores pueden almacenarse en registros si hay suficientes, se pueden utilizar instrucciones del estilo movimientos condicionales, etc. Por eso mismo, se puede observar una reducción significativa en el número de CSE de la búsqueda dicotómica desenrollada con respecto a la que se hizo con un bucle.

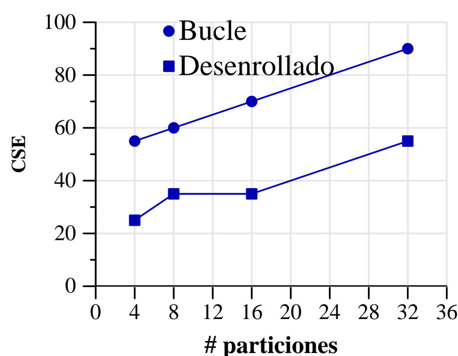

Figura 3.8: Particionado con el Counting Split para diferente número de particiones. El conjunto de datos que se particiona es de 2M datos de clave y puntero de 64 bits para un Power4 y una distribución Random.

A continuación se muestran los CSE para el particionado con Counting Split para tres distribuciones de datos diferentes, Random, S20 y D100. En las Figuras 3.9 y 3.10 se muestran los CSE obtenidos para Counting Split, realizando 32 particiones para datos de clave y puntero de 32 (Figura 3.9) y 64 (Figura 3.10) bits. Los resultados de ejecuciones reales se muestran con líneas continuas y marca. Con líneas discontinuas se muestran los CSE, según el modelo desarrollado, para el particionado de una distribución Random.

En las gráficas de la izquierda de las Figuras se muestran resultados para diferentes números de clave y puntero a ordenar. Aunque no se pueden distinguir, los CSE según el modelo coinciden con los CSE del particionado con Counting Split de distribuciones Random y S20 (líneas continuas que están en los 80 CSE en las Figuras).

En las gráficas de la derecha de las Figuras se muestran los CSE destinados a los subpasos del Paso (4) del algoritmo, que corresponden con los pasos del algoritmo

de conteo. En esas gráficas se muestran CSE para 4M (32 bits) y 2M (64 bits) datos a ordenar. El número de CSE del paso de conteo de Counting Split tiene un peso

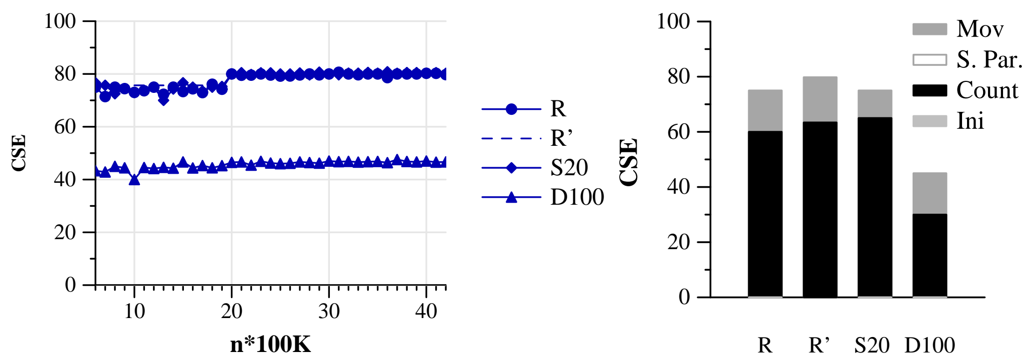

Figura 3.9: CSE de la ejecución real (continua) y del modelo (discontinua) para Counting Split para distribuciones Random, S20 y D100, de 100K a 4M elementos de 32 bits (izquierda). CSE para 4M datos, divididos entre los diferentes subpasos del Paso (4) de Counting Split (derecha).

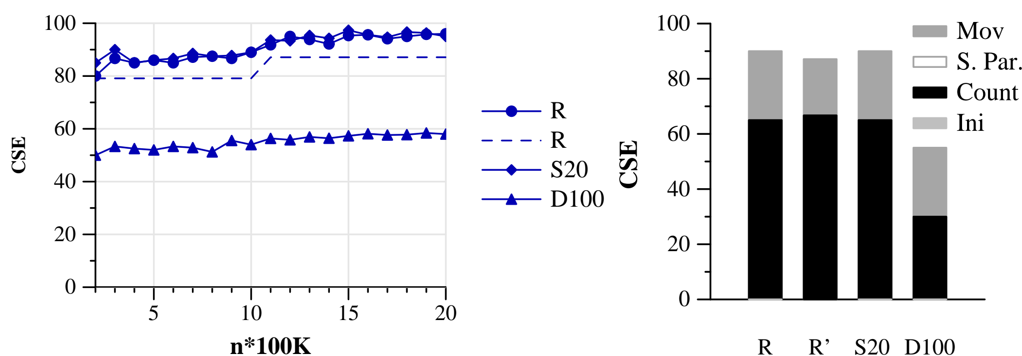

Figura 3.10: CSE de la ejecución real (continua) y del modelo (discontinua) para Counting Split para distribuciones Random, S20 y D100, de 100K a 2M elementos de 64 bits (izquierda). CSE para 2M datos, divididos entre los diferentes subpasos del Paso (4) de Counting Split (derecha).

significativo en todos los resultados obtenidos. Esto es debido a los ciclos gastados en las comparaciones de la búsqueda dicotómica.

En ambas Figuras 3.9 y 3.10, a partir de un cierto número de elementos a ordenar, se observa un aumento del número de CSE. Según el modelo, ese aumento se produce cuando las estructuras necesarias para la ordenación ya no caben en el tercer nivel de cache, es decir, cuando vamos a ordenar más de 2M y 1M datos para 32 y 64 bits

	Power4			
	32 bits		64 bits	
Fallos y Accesos $/n$	C	M	C	M
F. Primer nivel	0,012	0,021	0,023	0,067
A. Segundo nivel	0,058	0,069	0,116	0,137
A. Tercer nivel	0,001	0,002	0,003	0,004
A. Mem. principal	0,008	0,008	0,016	0,015
Modelo A. Mem	0,008	0,009	0,016	0,017
F. TLB	0,002	0,005	0,04	0,010
Modelo F. TLB	0,002	0,004	0,004	0,008

Tabla 3.2: Fallos de acceso al primer nivel de cache y de traducción del TLB, y número de accesos a segundo y tercer nivel de memoria cache y memoria principal dividido entre el total de elementos a ordenar, 4M (32bits) y 2M (64bits) datos para una distribución Random. C y M son el paso de conteo (4.2) y de movimiento (4.4) respectivamente, del Paso (4) del algoritmo.

respectivamente. En la Tabla 3.2 se muestran los fallos de accesos a primer nivel de cache y TLB por elemento a ordenar. Además se muestran el número de accesos por elemento a ordenar al segundo, tercer nivel de memoria cache y a memoria principal. En la misma tabla se muestran los cálculos estimados a partir de nuestro modelo para los accesos a memoria principal y fallos de TLB por elemento a ordenar, que son los dos aspectos que pueden influir más significativamente en el aumento del número de CSE. Como se puede observar, el número de accesos de memoria y fallos de TLB según el modelo se parecen mucho a los resultados, incluidos los fallos de TLB para el paso de movimiento. Esto es porque ahora el número de particiones que se realiza es mucho menor que el número de entradas de TLB y, por consiguiente, no hay apenas fallos por conflicto. Por lo tanto, el número de fallos de TLB es muy bajo. Esto es debido a que se ha conseguido particionar el conjunto de datos explotando la jerarquía de memoria. Entre otros aspectos, la escritura en el vector destino no provoca ahora un número elevado de fallos de TLB.

También se quiere hacer notar que tanto para datos de 32 como 64 bits hay un decremento de CSE para el paso de conteo, mostrado en las gráficas de la derecha de las Figuras, cuando la distribución de datos a ordenar es de todos duplicados (D100), en comparación con el resto de distribuciones de datos analizados. La explicación está en la reducción del número de fallos de predicción de saltos con respecto a otras distribuciones. En el caso que todos los datos sean duplicados, las comparaciones de la búsqueda dicotómica que son ciertas, son siempre las mismas, por lo tanto, el predictor

	Power4		
	R	S20	D100
Fallos Saltos	4,02	4,02	0,00
Coste Ponderado	48,02	48,02	0,00

Tabla 3.3: Fallos totales en predicción de saltos dividido entre el número total de elementos a ordenar (2M datos de 64 bits) para el Counting Split. Resultados para las distribuciones de datos Random (R), S20 y D100. En la fila inferior se ha ponderado el número de fallos por el número de ciclos de penalización por mala predicción, 12 ciclos para Power4.

de saltos sabe capturarlas y acierta siempre. En la Tabla 3.3 se muestra el número de fallos de predicción de saltos para los resultados obtenidos para el particionado de 2M de datos para las distribuciones Random (R), S20 y todo duplicados (D100). Teniendo en cuenta que la penalización por fallo en un Power4 es de al menos 12 ciclos, la diferencia en CSE entre distribuciones de datos es notable. En la misma Tabla se muestran los ciclos mínimos de penalización por fallos de predicción que se tienen. Eso no quiere decir que todos ellos se observen en el rendimiento final del particionado.

Por otro lado, analizando las particiones obtenidas para 2M datos de clave y puntero de 64 bits, la repartición de los elementos entre las particiones que se han obtenido permite explotar la localidad temporal de los datos de cada partición utilizando Radix sort. En el caso de de la distribución de datos D100, la distribución que se consigue no es equitativa. Con esta distribución de datos, todas las claves (repetidas) van a parar a una única partición, pero ésta no se tiene que ordenar.

3.4.2. Algoritmo Paralelo

El algoritmo paralelo de particionado con Counting Split (Algoritmo 9) realiza los siguientes pasos:

1. **Paso (1) - Comunicación de datos de muestreo:** Cada procesador realiza un muestreo de q/P claves de sus datos locales y comunica estas q/P claves al procesador P_0 (línea 2 del Algoritmo). Para calcular q, se utiliza la fórmula explicada en [13]. Sin embargo, aquí se limita el valor de q con tal de evitar comunicación innecesaria. En la sección de evaluación del algoritmo, más adelante, se analizará en detalle qué q se elige. El procesador P_0 crea el vector de muestreo con q claves que consisten en las $(P-1)q/P$ claves recibidas del resto de procesadores y q/P claves locales.

2. **Paso (2) - Ordenación del vector de muestreo:** El procesador P_0 ordena el vector de muestreo con Radix sort (línea 4 del Algoritmo).

3. **Paso (3) - Comunicación del vector de separadores:** El procesador P_0 crea el vector de separadores de $s-1$ claves tal y como se explica para el algoritmo secuencial de Counting Split. Después, P_0 comunica el vector de separadores al resto de procesadores con una comunicación de broadcast. De esta forma, cada procesador P_{pid} tendrá el vector de separadores local ($vector_separadores_{P_i}$). La elección de s está limitada por E_{TLB}, tal y como comentamos para el algoritmo secuencial. Todo este paso se realiza en las líneas 6 y 7 del Algoritmo 9.

4. **Paso (4) - Counting Split local:** Cada procesador aplica Counting Split a sus datos locales usando $vector_separadores_{P_{pid}}$. El algoritmo es el mismo que se vio en la sección de Counting Split secuencial, por lo que después de realizar este paso cada procesador tiene los datos divididos en $2s$ particiones (s particiones de sólo duplicados (impar) y s particiones de no todos duplicados (par)). Este paso se realiza en la línea 9 del Algoritmo.

5. **Paso (5) - Comunicación de contadores:** Los procesadores se comunican los $2s$ contadores utilizados en el particionado local con Counting Split que se ha realizado. De esta forma, cada procesador conoce cuantos elementos hay para cada una de las particiones globales. Este proceso se realiza en la línea 11 del Algoritmo.

6. **Paso (6) - Cálculo de la distribución de particiones:** Cada procesador, con los contadores del resto de procesadores y los locales, calcula el número de datos por partición global y cuál debe ser la división de las particiones para que haya equilibrio de carga. De esta forma, cada procesador conoce qué particiones están entre dos o más procesadores, las cuales deberían ser particionadas otra vez para conseguir equilibrio de carga. Estas particiones que se deberían particionar otra vez son las particiones frontera. Sólo las particiones frontera ($part_f$ en el Algoritmo 9) que no contienen sólo duplicados tienen que ser particionadas con Counting Split. Las particiones frontera que están formadas sólo por duplicados se pueden particionar sencillamente asignando un tanto por ciento de los duplicados a cada uno de los procesadores que comparten esa partición frontera. En la Figura 3.11, la partición B_1 es una partición frontera que está entre los Procesadores P_0 y P_1. Todo el proceso de cálculo de distribuciones se realiza en la línea 13 del Algoritmo.

   Las particiones frontera se particionan en s' subparticiones. El parámetro s' tiene las mismas limitaciones que s. Aquí, se les ha dado el mismo valor aunque s y s' podrían ser diferentes. En la Figura 3.11 se muestra la idea de esta nueva subdivisión de las particiones frontera con tal de conseguir un mejor equilibrio

de carga. Los procesadores, tras hacer el particionado local de sus datos (1 en la Figura), evalúan qué correspondencia debería haber entre procesadores y particiones (2 en la Figura) y realizan un particionado local (3 en la Figura) de las particiones que van a estar entre dos o más procesadores. Finalmente, cada procesador comunica las particiones o subparticiones a los procesadores que les corresponda (4 en la Figura). Los pasos a seguir para poder realizar una buena división de las particiones frontera son los Pasos (7) a (11), que se explican a continuación.

7. **Paso (7) - Comunicación de datos de muestreo:** Cada procesador realiza un muestreo de q'/P claves de su parte local de la partición frontera *part_f* y se los envía al procesador P_0 (Línea 17 del Algoritmo). q' se ha elegido más pequeño que q ya que esperamos que el número de elementos para la partición *part_f* sea menor que para todo el conjunto de elementos a ordenar (para el que se eligió q). El procesador P_0 recibe el muestreo de cada procesador P_i. Sin embargo, sólo toma un tanto por ciento de los muestreos recibidos de cada procesador P_i para crear el vector de muestreos para particionar la partición frontera *part_f*. Este tanto por ciento es proporcional a la contribución del procesador P_i a la partición frontera *part_f* global. Con ello evitamos que procesadores con una poca aportación en esa partición, puedan distorsionar la representatividad del vector de muestreo creado. En total, el vector del muestreo $vector_muestreo_{(P_0,part_f)}$ está formado por q'/P claves.

8. **Paso (8) - Ordenación de los datos del muestreo:** El procesador P_0 ordena los q'/P datos del muestreo con Radix sort (Línea 19 del Algoritmo).

9. **Paso (9) - Comunicación del vector de separadores:** El procesador P_0 crea el vector de separadores con $s' - 1$ claves y lo comunica al resto de procesadores con una comunicación broadcast (Líneas 21 y 22 del Algoritmo).

10. **Paso (10) - Counting Split local:** Cada procesador vuelve a aplicar Counting Split sobre los datos locales de la partición frontera *part_f* (Línea 24 del Algoritmo).

11. **Paso (11) - Comunicación de contadores:** Los procesadores se comunican los s' contadores de Counting Split local que han realizado (Línea 26 del Algoritmo). Nótese que son s' y no $2s'$. Para las particiones frontera no se distingue entre particiones con sólo duplicados y otras particiones. Se espera encontrar muchos duplicados en las particiones y además, de esta forma, se reduce la cantidad de contadores a comunicar.

12. **Paso (12) - Renombramiento de Procesadores:** Una vez se sabe cuantas particiones y subconjuntos de particiones frontera le deben corresponder a cada

procesador, se realiza un renombramiento de procesadores lógicos a físicos. Este paso es el mismo que se realiza para el particionado con Reverse Sorting paralelo. Este proceso se realiza en la línea 30 del Algoritmo.

13. **Paso (13) - Comunicación de datos:** Por un lado, cada procesador comunica las particiones locales que tienen que ir a otros procesadores. Por otro lado, recibe las particiones locales del cual el procesador es destino. Las particiones que son locales a un procesador y le correspondan a este procesador, simplemente no se comunican (Línea 32 del Algoritmo).

Resumiendo el algoritmo, se puede decir que con Counting Split secuencial, explicado en la sección anterior y que se aplica en el algoritmo paralelo, las particiones creadas, durante el particionado local, deberían tener un tamaño similar o bien esta formadas por claves repetidas. Por esto mismo, es factible obtener una buena distribución de los datos entre los procesadores. Sin embargo, si no se consiguiera esa distribución equilibrada de los datos, ya que q y s se han limitado, las particiones frontera se pueden particionar otra vez tal y como se explica en los pasos del Algoritmo 9.

En el caso de que la partición frontera sea una partición impar (sólo duplicados), la decisión de qué hacer depende de cómo se quiera que el vector ordenado quede distribuido entre los procesadores. Por un lado, podemos no comunicarlas ya que esas particiones no tienen que ser ordenadas. Por consiguiente, los procesadores podrían tener porciones muy desiguales del vector ordenado. Esto no significa que se tenga desequilibrio de carga durante la ordenación. Por otro lado, podemos comunicar la parte que toque a cada procesador. Si una repartición de duplicados está en la frontera de dos procesadores, la partición es inmediata ya que se puede decidir cortar por cualquier posición indicando el tanto por ciento de los datos que le corresponden a cada procesador.

Evaluación del Algoritmo

Las Figuras 3.12, y 3.13 muestran los CSE del particionado con Counting Split paralelo, variando el parámetro q (para $256P$ (1), $512P$ (2), $1024P$ (3) y $2048P$ (4) muestreos) para un número total de 4M datos (gráfica de arriba) y un número fijo de 4M datos por procesador (4PM datos, gráfica de abajo), variando el número de procesadores de 2 a 16. Las Figuras muestran los CSE para distribuciones de datos Random y D50 para claves y punteros de 64 bits. Los CSE se muestran descompuestos según diferentes partes del algoritmo. De arriba a abajo se muestran CSE de preparación de los mensajes (prepare), CSE acumulados de los Pasos (7) a (11) del Algoritmo 9 (llamado adicional en las Figuras), CSE para la comunicación de datos (*AlltoAll*, Paso (13) del Algoritmo), CSE para la comunicación de los contadores (Paso (5), en la Figura Transpose), CSE para el particionado con Counting Split local, CSE de la

Algoritmo 9: Counting Split Paralelo

1: – **Paso (1):** Comunicación datos muestreo
2: $vector_muestreo_{P_0} \leftarrow envia_muestreo(\forall i P_i, q/P)$

3: – **Paso (2):** Ordenación del vector muestreo
4: $Radix(vector_muestreo_{P_0}, q)$

5: – **Paso (3):** Broadcast *vector_separadores*
6: $vector_separadores_{P_0} \leftarrow obtener_separador(vector_muestreo_{P_0}), s)$
7: $vector_separadores_{P_{pid}} \leftarrow broadcast(vector_separadores_{P_0})$

8: – **Paso (4):** Counting Split local
9: $Counting_Split(S, n, vector_separadores_{P_{pid}}, s)$

10: – **Paso (5):** Comunicación de contadores
11: $GlC \leftarrow allgather(C, 2s)$

12: – **Paso (6):** Cálculo de distribución de particiones
13: $distribucion_particiones(GlC)$

14: **para cada** *particion_frontera part_f* **hacer**
15: **si** *part_f* <> *duplicados* **entonces**
16: – **Paso (7):** Comunicación muestreo partición
17: $vector_muestreo_{(P_0, part_f)} \leftarrow envia_muestreo(\forall i P_i, q'/P)$

18: – **Paso (8):** Ordenación $vector_separadores(part_f)$
19: $Radix(vector_muestreo_{(P_0, part_f)}, q'')$

20: – **Paso (9):** Broadcast $vector_separadores(part_f)$
21: $vector_separadores_{P_0, part_f} \leftarrow obtener_separador(vector_muestreo_{P_0, part_f}), s')$
22: $vector_separadores_{P_{pid}, part_f} \leftarrow broadcast(vector_separadores_{P_0, part_f})$

23: – **Paso (10):** Counting Split local para partición
24: $Counting_Split(S, n, vector_separadores_{(P_{pid}, part_f)}, s')$

25: – **Paso (11):** Comunicación de contadores para partición
26: $GlC \leftarrow allgather(C, s')$
27: **fin si**
28: **fin para**
29: – **Paso (12):** Renombramiento de Procesadores
30: $renombre \leftarrow renombramiento(GlC)$

31: – **Paso (13):** Comunicación de datos
32: $comunicacion_datos(S, D, GlC, renombre)$

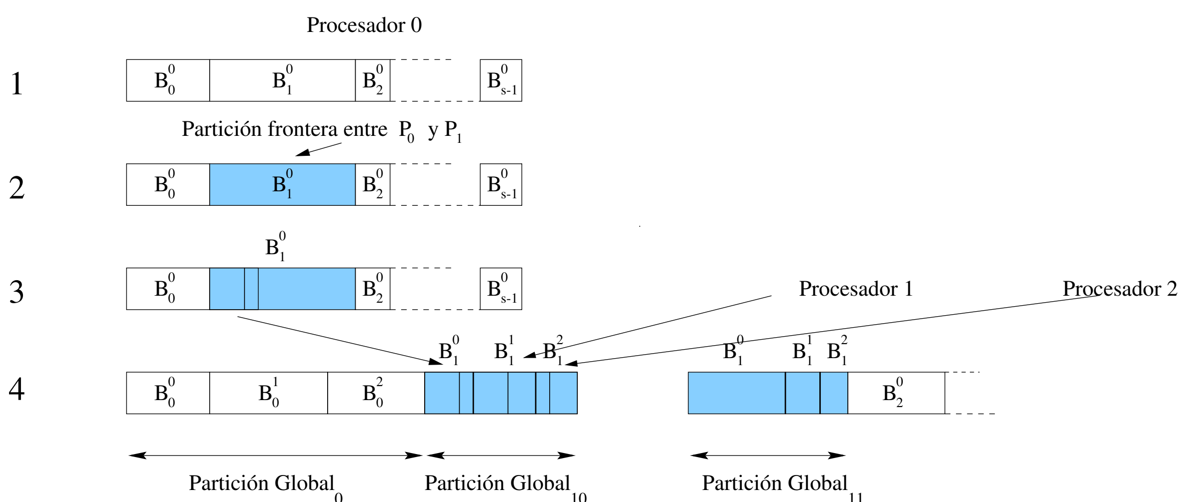

Figura 3.11: Repartición de una partición frontera entre dos procesadores.

comunicación de broadcast del vector de separadores (Paso (4), broadcast Split en la Figura) y finalmente, CSE para la ordenación y comunicación del muestreo, Pasos (2) y (1) del Algoritmo 9 respectivamente, mostrados como Radix y Com. Muestreo en la Figura.

En las Figuras sólo se muestran los CSE de las ejecuciones hasta una $q = 2048P$ ya que según el modelo, detallado en el Apéndice C, se observó que los CSE para comunicación y ordenación de los muestreos, para un mayor número de muestreos, empieza a aumentar, sin que por ello se consiga mejor equilibrio de carga entre los procesadores. Las líneas discontinuas mostradas en las Figuras son los CSE totales según el modelo, desarrollado en el Apéndice C. Se puede ver que la tendencia es la misma y que para q mayores que $2048P$ los CSE aumentan dependiendo del número de datos a ordenar por procesador y del número de procesadores. Así, de las Figuras se puede deducir lo siguiente:

- El número de CSE invertido en obtener el muestreo es despreciable tanto ordenando un número fijo de datos con todos los procesadores como cuando se ordena un número de elementos fijo por procesador.

- El número de CSE de los Pasos (7) a (11) del Algoritmo (adicional en la Figura) para subdividir las particiones frontera es apreciable cuando el número de elementos por procesador es pequeño. Además, este coste es más significativo en la distribución Random ya que las particiones que se tienen que subdividir por ser particiones frontera no son de duplicados. En cambio, para la D50, hay una partición que se tiene que subdividir entre varios procesadores pero es de

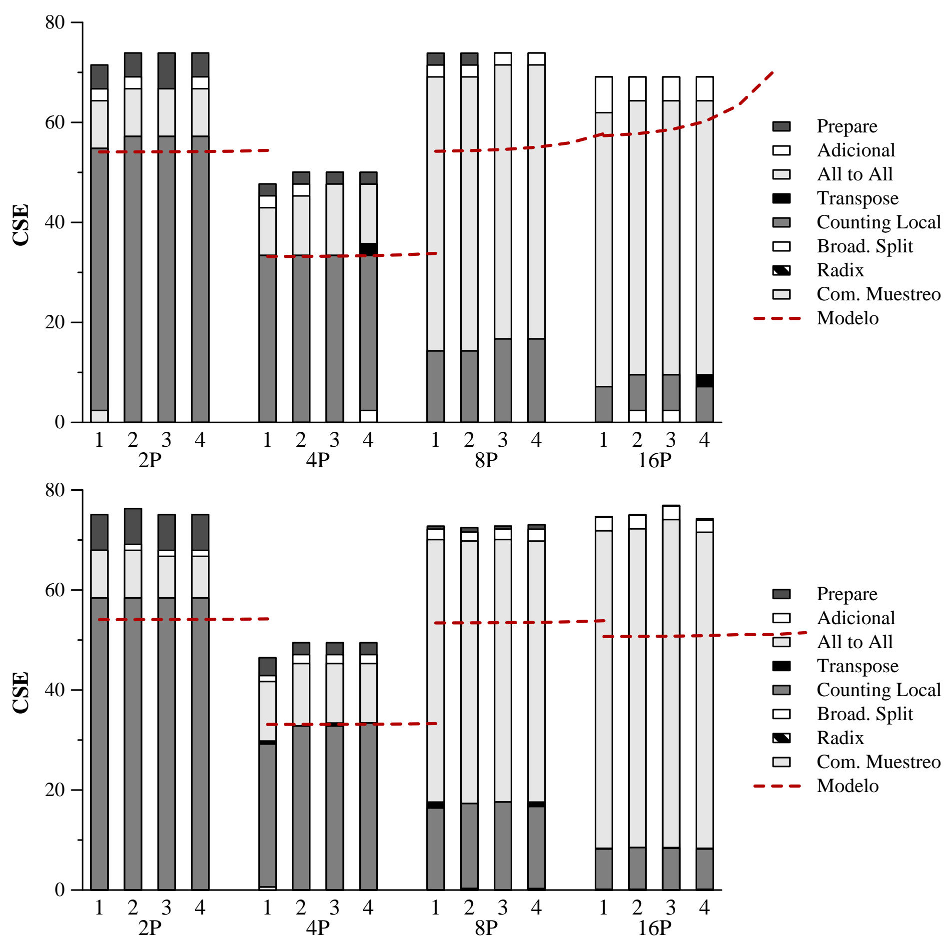

Figura 3.12: CSE del particionado con Counting Split variando q ($256P$ (1), $512P$(2), $1024P$(3) y $2048P$(4)) y P (de 2 a 16) para 4M (arriba) y $4P$M (abajo) datos de clave y puntero a ordenar. Los resultados son para una distribución Random. Las líneas discontinuas muestran CSE totales según el modelo para las mismas q que en la ejecución real y también para $q = 4096P$ y $q = 8192P$.

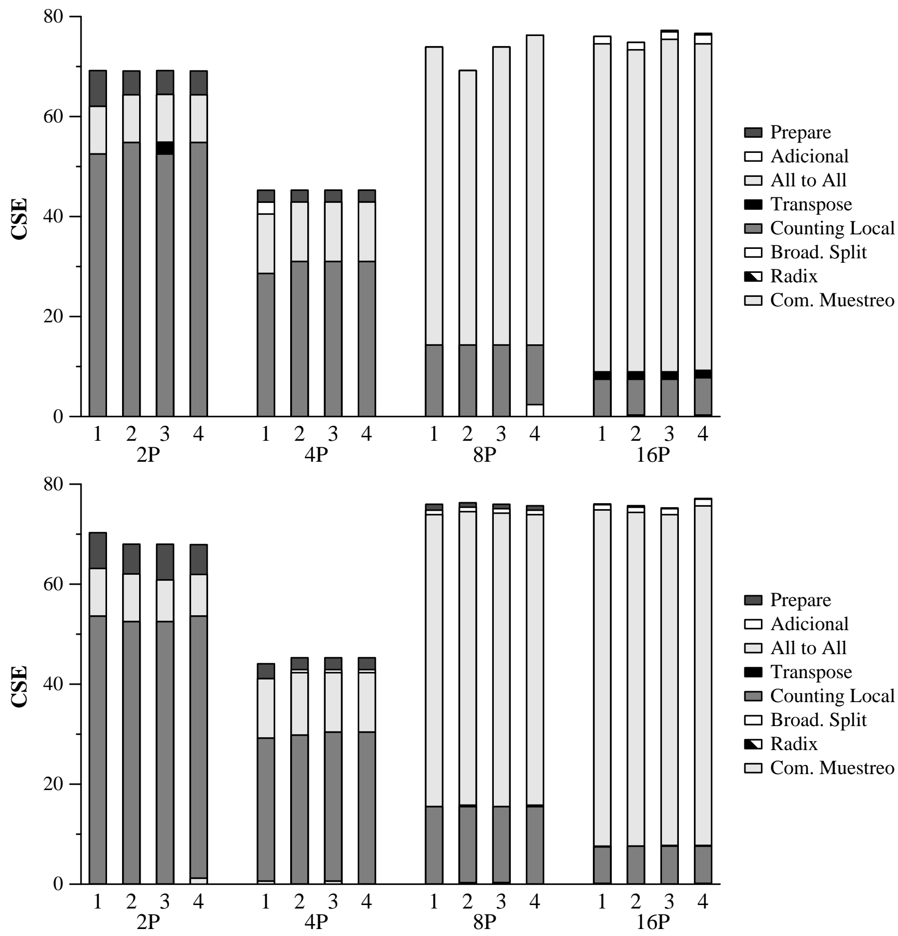

Figura 3.13: El mismo análisis que en la Figura 3.12 pero para una distribución D50.

duplicados, por lo que se puede hacer dando un tanto por ciento a cada uno de los procesadores que comparten esa partición.

- La probabilidad de tener que subdividir particiones frontera es mayor cuando el número de muestreos es menor. $q = 2048P$ permite obtener un buen equilibrio de carga sin pagar muchos CSE para los Pasos (7) a (11), del reparticionado de particiones frontera.

- Hay un aumento en el número de CSE cuando se pasa de 4 a 8 procesadores, tal y como se vio para Reverse Sorting paralelo. El ancho de banda de comunicación efectivo es mucho menor cuando se pasa a 8 procesadores, al tener que hacer comunicación entre nodos.

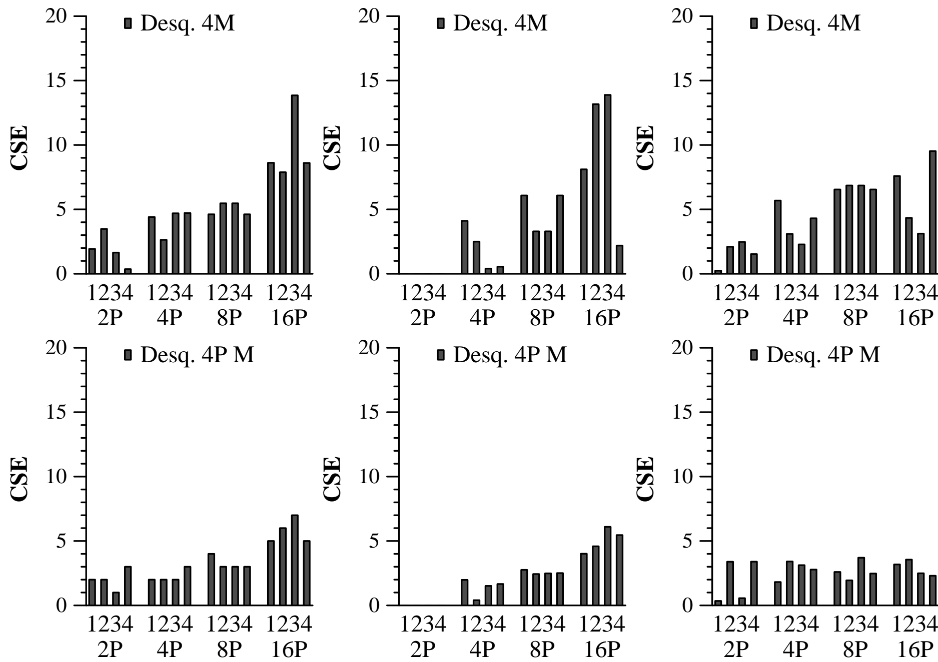

Figura 3.14: Desequilibrio de carga obtenido con Counting Split variando q ($256P$(1), $512P$(2), $1024P$(3) y $2048P$(4)) y P (de 2 a 16) para 2M (arriba) y $2P$M (abajo) de datos de 64 bits. De izquierda a derecha, corresponden al particionado de datos con distribución Random, D50 y Gaussian.

Finalmente, en la Figura 3.14 se muestran los desequilibrios de carga obtenidos en las ejecuciones con la técnica de particionado de Counting Split. En las gráficas se ve el desequilibrio de carga obtenido para las distribuciones de datos Random, Gaussian y D50, variando el número de muestreos a realizar. Cuando el número de

elementos a ordenar por procesador es fijo y el número de samples es de $q = 2048P$, el tanto por ciento de desequilibrio de carga obtenido no supera el 5,5 % del número de elementos que le corresponde a cada procesador (Figura 3.14, arriba). Sin embargo, el desequilibrio de carga, cuando el número de elementos a ordenar es fijo aunque se varíe el número de procesadores, es significativo. Este desequilibrio de carga es mayor cuando el número de procesadores aumenta. Las causas son las mismas que indicamos para Reverse Sorting paralelo:

1. El número de particiones que se realiza, 64, es pequeño en relación al número de procesadores que va aumentando.

2. Se permite un desequilibrio de carga que puede hacer que procesadores se queden sin elementos a ordenar. Si no se permitiera ningún tipo de desequilibrio de carga es probable que se tuviera que pagar con un alto número de CSE en los Pasos 7 a 11 del Algoritmo, para reparticionar particiones frontera hasta obtener un equilibrio de carga perfecto.

3.5. Mutant Reverse Sorting

Esta técnica se basa en las dos técnicas de particionado desarrolladas y estudiadas en esta tesis, Reverse Sorting y Counting Split.

Como se puede deducir del análisis realizado hasta el momento, la elección del algoritmo de particionado que obtenga una buena distribución de los datos a un bajo coste no es sencillo.

Por un lado, como se ha visto, el coste de una iteración de Reverse Sorting es bajo. Sin embargo, una única iteración no asegura una repartición equitativa de los datos entre los subconjuntos cuando se tienen datos con sesgo o con duplicados. En ese caso se tiene que hacer más de una iteración de Reverse Sorting. Por consiguiente, el número final de CSE particionando con Reverse Sorting puede ser alto, véase las Figuras 3.1, para el algoritmo secuencial y Figuras 3.5 para el algoritmo paralelo de Reverse Sorting cuando se hacen 3 iteraciones o más de Reverse Sorting.

Por otro lado, Counting Split ayuda a la obtención de una buena repartición de los datos en conjuntos de tamaño similar a un coste aceptable, sea cual sea la distribución de datos, y en su defecto, se consiguen particiones de un número pequeño de datos a ordenar o un conjunto de datos de duplicados que no requieren ser ordenados. Sin embargo el coste de un paso de Counting Split es mayor que el de una iteración de Reverse Sorting. Por lo que si el número de iteraciones de Reverse Sorting necesarias no es elevado, conviene utilizar Reverse Sorting.

Así, lo que se propone es:

Usar la técnica de Reverse Sorting cuando se tenga que ordenar una distribución de datos sin o con muy poco sesgo en los datos, ya que el número de iteraciones de

Reverse Sorting será pequeño. En cualquier otro caso, se usará la técnica de Counting Split.

Elección de Reverse Sorting o Counting Split

Así, se aplicará Reverse Sorting a no ser que sea preciso realizar un número de iteraciones de Reverse Sorting muy elevado; en cuyo caso se decidirá aplicar Counting Split. Es por ello que debe estimarse el número de iteraciones de Reverse Sorting que serían necesarias para particionar un conjunto de datos dado con tal de que:

1. Secuencial: se pueda explotar la jerarquía de memoria en la ordenación de cada partición de datos obtenida en el particionado.

2. Paralelo: se pueda obtener un particionado de los datos que permita una ordenación con poca comunicación y, en la medida de lo posible, consiga un buen equilibrio de carga sin con ello penalizar el rendimiento final.

A continuación se propone una técnica para decidir entre una u otra técnica de particionado, Reverse Sorting o Counting Split, tanto para el algoritmo secuencial como el paralelo. A esta técnica se le llama Mutant Reverse Sorting. Más adelante se explicará el porqué de este nombre.

En esta sección se explicará el algoritmo de la técnica combinada y se mostrará una comparativa en CSE de las técnicas de particionado Reverse Sorting, Counting Split y Mutant Reverse Sorting.

3.5.1. Algoritmo Secuencial

Tal y como se comentó, esta técnica decide entre Reverse Sorting y Counting Split basándose en una estimación del número de iteraciones de Reverse Sorting necesarias. Esta técnica, que se propone en esta tesis, se muestra en el Algoritmo 10.

El Algoritmo 10 sigue los siguientes pasos:

1. **Paso (1) - Obtención de un vector de muestreo:** Se crea un vector de muestreo con q claves, obtenidas aleatoriamente del conjunto a ordenar tal y como se hace con Counting Split secuencial (Línea 2 del Algoritmo 10). Estas q claves son, en total, un T_q tanto por ciento de todas las claves del conjunto a ordenar. Además, se inicializará a cero el número de claves a las que se ha aplicado Reverse Sorting (N_{rev}) (Línea 3 del Algoritmo 10).

2. **Paso (2) - Simulación de Reverse Sorting**. Este paso se implementa de forma recursiva y estima el número de iteraciones que se aplicaría Reverse Sorting a todo el conjunto de datos (Línea 5 del Algoritmo 10). Para poder estimarlo, se aplica Reverse Sorting al conjunto de q claves extraídas aleatoriamente. Si el

número de iteraciones necesarias para poder obtener una buena repartición de estas q claves es más grande que $\frac{CSE_{sec_cs}}{CSE_{sec_rev}}$ entonces se aplicará Counting Split al conjunto de datos en vez de Reverse Sorting. Para realizar esta estimación se llama a la función *Simula Reverse* en la línea 5 del Algoritmo 10, con las q claves. El Algoritmo 11 describe esta función y sus pasos son:

- **Paso (2.1) - Iteración de Reverse Sorting (si es necesario):** A N_{rev} se le añade el número de claves sobre la cual se está haciendo este paso de Reverse Sorting. Si $N_{rev} > \frac{CSE_{sec_cs}}{CSE_{sec_rev}}q$, esto quiere decir que es probable que el número de CSE al aplicar la técnica de Reverse Sorting sea mayor que los CSE de aplicar la técnica de Counting Split. Esto es porque se estima que se haría un número de iteraciones de Reverse Sorting que igualaría los CSE de aplicar Counting Split. En este caso se para (Líneas 3 a 5 del Algoritmo 11).

  En otro caso, se aplica una iteración de Reverse Sorting al vector de n' claves que pasaron en la llamada a la función (Línea 6 del Algoritmo 11).

- **Paso (2.2) - Comprobación de si hay particiones grandes:** Una vez creadas las particiones del vector de n' claves, se tiene que mirar si es necesario particionar más. Si se encuentra una partición con n_i elementos donde $n_i > N_{grande}$, se llama, recursivamente, al Algoritmo 11 para el vector de estos n_i elementos.

  El parámetro $N_{grande} = \frac{T_q}{100} min(R_{cache_i}, R_{TLB})$ es el T_q % del mínimo entre el número de claves que caben en el nivel de memoria cache que queremos explotar (R_{cache_i} elementos) y la memoria que puede mapear la estructura de TLB expresada en elementos (R_{TLB} elementos).

  Con ello se está suponiendo que, en caso de estar ordenando todo el conjunto de datos, la partición que se hubiera obtenido para el mismo rango de valores, que la obtenida para las n_i claves, sería también demasiado grande para explotar el mismo nivel de memoria cache y/o la estructura del TLB.

  Al final de este algoritmo, N_{rev} tiene acumulado el número de claves sobre los que se ha aplicado una o más iteraciones de Reverse Sorting.

3. **Paso (3) - Elección de una u otra técnica de particionado:** Si $N_{rev} > \frac{CSE_{sec_cs}}{CSE_{sec_rev}}q$, se decidirá usar el particionado con Counting Split ya que en este caso el número de iteraciones de Reverse Sorting que se necesitarían sobre todo el conjunto de datos, equivalente al número efectuado para las q claves elegidas, haría que el número de CSE fuera mayor al de aplicar Counting Split.

   La relación anterior se debe ajustar para cada procesador en cuestión. Esto se puede obtener con un simple experimento donde se determine el número de CSE para una iteración de Reverse Sorting y para Counting Split.

En el caso de elegir particionar con Counting Split se realizará un número de particiones proporcional al número de elementos a ordenar y no un número fijo de particiones como se hacía cuando se propuso Counting Split para datos de 64 bits.

Todo este proceso de elección se realiza en las líneas 7 a 11 del Algoritmo 10.

Algoritmo 10: Mutant Reverse Sorting(S, n, b_r)

1: – **Paso (1):** Obtención de un Muestreo
2: $vector_muestreo \leftarrow muestreo_aleatorio(S, q)$
3: $N_{rev} \leftarrow 0$

4: – **Paso (2):** Simulación de Reverse Sorting
5: $Simular_Reverse(S, q, b, b_r, N_{rev})$

6: – **Paso (3):** Elección
7: **si** $N_{rev} > \frac{C_{split}}{C_{rev}} q$ **entonces**
8: $Counting_Split$
9: **sino**
10: $Reverse_Sorting$
11: **fin si**

Evaluación del Algoritmo

Las Figuras 3.15 y 3.16 muestran los CSE de particionar con Reverse Sorting, Counting Split y Mutant Reverse Sorting para datos de 32 y 64 bits respectivamente. Se muestran los resultados para distribuciones de datos Random, S20, S40 y D100. En todas las gráficas mostradas, Mutant Reverse Sorting se muestra con línea continua y una marca en forma de rombo. Reverse Sorting se muestra con una línea continua y Counting Split en línea discontinua, ambos sin marca. Como se puede observar, Mutant Reverse Sorting tiene un número de CSE similar a la mejor de las otras dos técnicas que combina.

Para claves de 32 bits, se puede observar que para las distribuciones Random, S20 y S40, aplicar Reverse Sorting es mejor que aplicar Counting Split. Para una distribución Random, sólo es necesario realizar una iteración de Reverse Sorting, lo cual es menos costoso que aplicar Counting Split. En el caso de la distribución S20, se precisan dos iteraciones de Reverse Sorting a partir de un determinado número de datos a ordenar tal y como se explicó en la Sección de Reverse Sorting de este Capítulo. Sin embargo, aunque S20 y S40 requieren de dos iteraciones de Reverse Sorting, el número de CSE en estas dos iteraciones es menor que un paso de Counting Split.

Algoritmo 11: Simula_Reverse (S, n', b, b_r, N_{rev})

1: – **Paso (2.1):** Aplicar paso de Reverse Sorting
2: $N_{rev} \leftarrow N_{rev} + n'$
3: **si** $N_{rev} > \frac{C_{split}}{C_{rev}} q$ **entonces**
4: Parar.
5: **fin si**

6: $< D, C > \leftarrow particiona_conteo(S, n', b, b_{m-1})$

7: – **Paso (2.2):** Particionado Recursivo de ciertas particiones
8: $off \leftarrow 0$
9: $nb \leftarrow 2^{b_r}$
10: **para** $ib = 0$ a $nb - 1$ **hacer**
11: **si** $((C[ib] > T_q R_{TLB}) \vee (C[ib] > T_q R_{cache_i})) \wedge (b - b_r > 0)$ **entonces**
12: $Simula_Reverse(D + off, S + off, C[ib], b - b_r)$
13: **fin si**
14: $off \leftarrow off + C[ib]$
15: **fin para**

Por el contrario, en el caso de una distribución D100, los CSE de aplicar un número elevado de iteraciones de Reverse Sorting hace que salga más a cuenta aplicar Counting Split. Además, el número de CSE del particionado con Counting Split de una distribución de todos duplicados (D100) es menor que para el resto de distribuciones estudiadas (véase la Sección de Counting Split en este Capítulo). Por consiguiente, aún sale más a cuenta.

Para claves de 64 bits y distribución Random, Reverse Sorting sigue siendo la mejor técnica de particionado. Mutant Reverse Sorting se decide por esta técnica de particionado. Sin embargo, se puede ver que Reverse Sorting tiene un coste elevado para S20, S40 y D100. Mutant Reverse Sorting decide aplicar Counting Split para estas distribuciones de datos, aunque se observa una diferencia en CSE entre Mutant Reverse Sorting y Counting Split. Esta diferencia se debe a que Counting Split siempre realiza un número fijo de particiones, cuando para conjuntos relativamente pequeños de datos se podría hacer el particionado en menos particiones. Si el número de particiones es menor, el coste de la búsqueda dicotómica se reduce.

Por lo tanto, ahora, para claves 64 bits, tener una distribución S20 o S40 significa que las claves tienen los 12 y 25 bits más significativos iguales respectivamente, el doble que para claves de 32 bits. Por consiguiente, se precisan más iteraciones de Reverse Sorting para conseguir un buen particionado. Además, cada iteración de Reverse Sorting es más costosa ya que se trabaja con un volumen mayor de datos.

En general, se observa que la diferencia de rendimiento puede ser notable según la elección que se hubiese hecho. Además, algo que no se observa en los resultados ya que no se muestran aquí los CSE de ordenación de particiones, es que cuando se decide por Counting Split una parte de los bits más significativos de la clave será la misma para todas las claves. Este número de bits comunes seguramente es mayor que el que tendríamos al hacer una iteración de Reverse Sorting (b_r bits comunes). En este caso, cuando se ordene con Radix sort se tendrá menos trabajo a realizar.

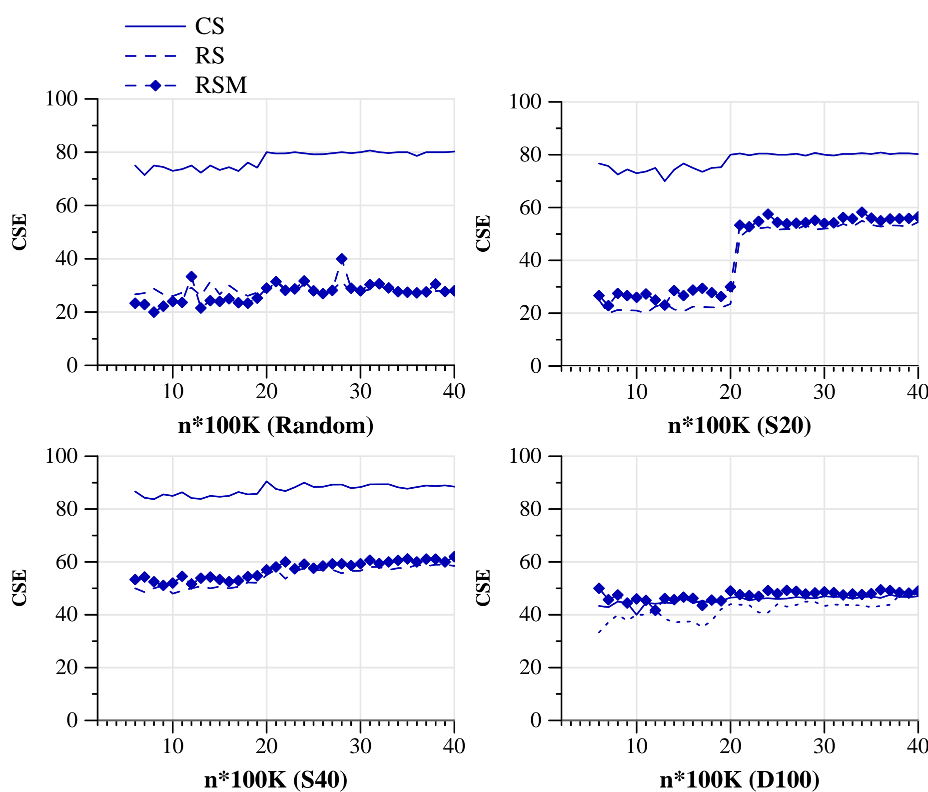

Figura 3.15: Comparativa en CSE del particionado de claves y punteros de 32 bits con Reverse Sorting (RS), Counting Split (CS) y Mutant Reverse Sorting (RSM). Se muestran CSE para una distribución Random, S20, S40 y D100.

3.5.2. Algoritmo Paralelo

El funcionamiento del particionado con Mutant Reverse Sorting paralelo es muy parecido al del algoritmo secuencial. Ahora, sin embargo, la decisión de realizar otra iteración de Reverse Sorting sobre los datos muestreados se basa en si se consigue o no equilibrio de carga de los elementos entre los procesadores. El Algoritmo 12 explica

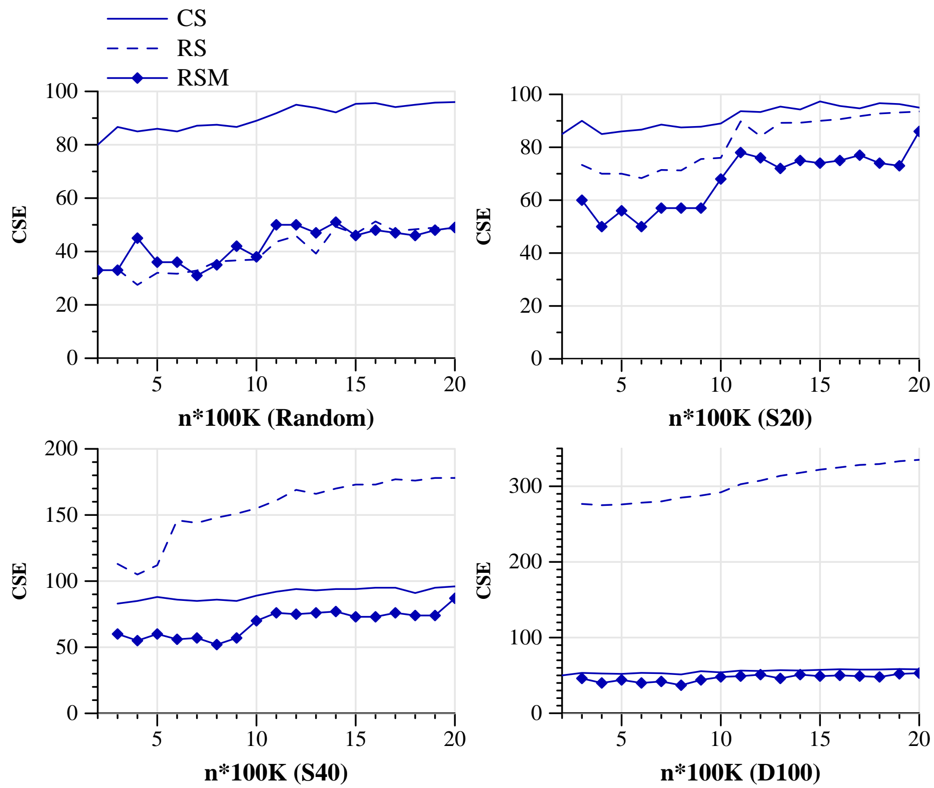

Figura 3.16: Comparativa en CSE del particionado de claves y punteros de 64 bits con Reverse Sorting (RS), Counting Split (CS) y Mutant Reverse Sorting (RSM). Se muestran CSE para una distribución Random, S20, S40 y D100.

la técnica de Reverse Sorting Paralelo. Éste sigue los siguientes pasos:

1. **Paso (1) - Obtención y comunicación del vector de muestreo:** Cada procesador realiza un muestreo de q/P claves de sus datos locales y comunica estas q/P claves al procesador P_0 (Línea 2 del Algoritmo 12). Se puede pensar que estas q claves representan bien la distribución de datos del conjunto a ordenar. Las q claves representan el T_q tanto por ciento del total de las claves del conjunto a ordenar.

2. **Paso (2) - Aplicar Reverse Sorting Local en P_0:** El procesador P_0 realiza una iteración de Reverse Sorting utilizando un dígito de $2b_r$ bits sobre el vector de muestreo que recibió en el Paso (1). Es decir, como si se aplicara una iteración de Reverse Sorting usando los b_r bits más significativos, y a todas las particiones obtenidas, se les aplicara otra iteración de Reverse Sorting por los siguientes b_r bits más significativos de las claves. Esto se hace en la línea 4 del Algoritmo.

3. **Paso (3) - Comprobación de equilibrio de carga y elección:** Si después de realizar el particionado no se puede encontrar un buen equilibrio de carga de las claves del muestreo entre los P procesadores, eso quiere decir que no se puede obtener un buen equilibrio de carga de todo el conjunto a ordenar, con dos iteraciones de Reverse Sorting. Un buen equilibrio de carga se consigue si se es capaz de distribuir equitativamente las q claves del muestreo entre los P procesadores con un desequilibrio de carga máximo de $T_q * U$ claves. U es el desequilibrio de carga permitido en el particionado con Reverse Sorting paralelo. T_q es el tanto por ciento del total de claves, que son las q claves del muestreo. Si no se consigue un buen equilibrio de carga después de este particionado, quiere decir que es probable que se necesiten más de dos iteraciones de Reverse Sorting para particionar todo el conjunto de datos. A la vista de los resultados obtenidos y de las estimaciones efectuadas con el modelo, más de dos iteraciones de Reverse Sorting paralelo suponen un número de CSE elevado y superior al de un particionado con Counting Split. En este caso se decidirá usar el particionado con Counting Split.

Nótese que sólo es preciso la comunicación de los muestreos para decidir particionar con Reverse Sorting o con Counting Split. En el caso de aplicar Counting Split posteriormente, el muestreo de los datos ya no es necesario.

Evaluación del Algoritmo

En la Figura 3.17 se comparan los CSE del particionado de $4PM$ datos con Reverse Sorting, Counting Split y Mutant Reverse Sorting paralelo para diferentes distribuciones de datos. En la gráfica de arriba se muestran los resultados para P=2 procesadores. En la gráfica de abajo se muestran los resultados para P=16 procesadores.

Algoritmo 12: Mutant Reverse Sorting Paralelo(S, n, b_r)

1: – **Paso (1):** Obtención y Comunicación del vector de muestreo
2: $vector_muestreo_{P_0} \leftarrow envia_muestreo(\forall i P_i, q/P)$

3: – **Paso (2):** Aplicar una iteración de Reverse Sorting doble
4: $< D, C > \leftarrow particiona_conteo(vector_muestreo_{P_0}, q, b, 2b_r)$

5: – **Paso (3):** Comprobación de equilibrio de carga y elección
6: **si** $equilibrio_carga(C)$ **entonces**
7: $Reverse_Sorting$
8: **sino**
9: $Counting_Split$
10: **fin si**

Como se puede observar, Mutant Reverse Sorting invierte un número de CSE similar al de la técnica de particionado más rápida. Para 2 procesadores (gráfica de arriba), Reverse Sorting paralelo es capaz de obtener un buen equilibrio de carga para las distribuciones Random, S20 y D50. En el caso de 16 procesadores, Reverse Sorting paralelo sólo consigue una buena distribución de los datos para las distribuciones Random y S20. Para el resto de distribuciones necesita realizar más de 3 iteraciones de Reverse Sorting.

Counting Split ofrece mejor o igual rendimiento que Reverse Sorting para todas las distribuciones (mostradas en las gráficas) a excepción de la distribución Random y la D50 para 2 Procesadores. Para la distribución Random, Reverse Sorting obtiene un buen equilibrio de carga con una única iteración. El caso de ordenar una distribución D50 con 2 procesadores es un caso especial. Con una única iteración de Reverse Sorting se consigue distribuir equitativamente los datos; un procesador se queda con la partición que tiene asignada la mitad de los elementos (con todos los duplicados) y el otro procesador, el resto de particiones.

En cualquier caso, la técnica de decisión explicada es capaz de capturar todas estas situaciones.

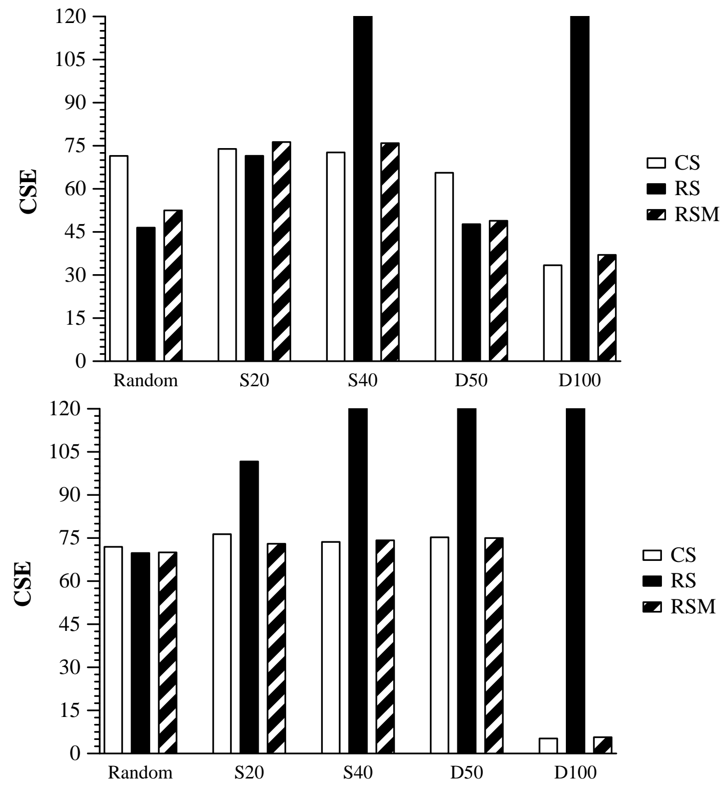

Figura 3.17: Comparativa en CSE del particionado de claves y punteros de 64 bits con Reverse Sorting (RS), Counting Split (CS) y Mutant Reverse Sorting (RSM). Se muestran CSE para el particionado de $4MP$ datos para P=2 (arriba) y P=16 (abajo) procesadores, para una distribución Random, S20, S40 y D100.