# Algoritmos Pararelos Adaptativos de Ordenamiento

David A. Koufaty †

## Resumen

En este trabajo se presentan algoritmos paralelos para el problema de ordenación, basados en algoritmos secuenciales adaptativos. Algunos de los algoritmos fueron implementados sobre una red de transputadores y se compara su rendimiento con otros algoritmos de ordenación secuencial.

Keywords: algoritmos paralelos, algoritmos de ordenamiento, algoritmos adaptativos de ordenamiento

## 1 Introducción

Se dice que un algoritmo de ordenamiento es adaptativo si ordena una secuencia en tiempo proporcional a la longitud y el desorden de la misma; es decir, si la entrada está relativamente ordenada el algoritmo debe ser considerablemente más rápido que cuando no lo está, llegando en el peor caso a orden $O(n \log n)$ para una secuencia de longitud $n$. Este tipo de algoritmo es de interés intrínseco ya que la identificación de casos "fáciles" llevaría a algoritmos muy eficientes.

Para medir el desorden de una secuencia dada se han propuesto numerosas métricas que definen el preordenamiento de una secuencia [Man85, Ski88], entre ellas el número de pares de elementos en orden incorrecto o inversiones (*Inv*), el número de subsecuencias contiguas ascendentes o corridas (*Runs*), el mínimo número de eliminaciones de elementos para dejar una subsecuencia ordenada (*Rem*), la máxima distancia de un elemento a su posición correcta en el ordenamiento (*Max*), el mínimo número de intercambios para ordenar la secuencia

†Departamento de Computación, Universidad Simón Bolívar, Apartado Postal 89000, Caracas 1080A, VENEZUELA

(*Exc*), el mínimo número de listas anidadas (*Enc*) y el mínimo número de subsecuencias monótonas (ascendentes o descendentes) requeridas para cubrir la secuencia (*SMS*).

En base a la definición formal de una métrica, en [Man85] se presenta el concepto de optimalidad de un algoritmo de ordenamiento con respecto a una métrica de preordenamiento (por ejemplo, ordenamiento por inserción local es óptimo con respecto a *Inv*, *Runs* y *Rem*). Por otra parte se ha demostrado que algunas métricas son algorítmicamente más finas, en el siguiente sentido: si $M_1$ es más fina que $M_2$, entonces todos los algoritmos optimales con respecto a $M_1$ lo son con respecto a $M_2$ (por ejemplo, un algoritmo optimal con respecto a *SMS* es optimal con respecto a *Runs* y *Max* y puede ser fácilmente modificado para ser optimal con respecto a *Rem* y *Exc*), por lo cual es más interesante encontrar algoritmos optimales con respecto a métricas muy finas.

En la literatura puede encontrarse varios algoritmos secuenciales adaptativos para ordenamiento. El objetivo de este trabajo consiste en presentar versiones paralelas de algunos de ellos, en particular: *Melsort* [Ski88], *Slabsort* [LP90] y *Randomized Slabsort* [ECW91]. Consideraremos además que el orden en el cual deben quedar los elementos de la secuencia es no–decreciente.

## 2 Partición de $S$

Los algoritmos paralelos que siguen requieren la solución paralela del siguiente problema: dada una secuencia cualquiera $S$ de tamaño $n$ distribuída uniformente entre $p$ procesadores ($S = S_1 S_2 \ldots S_p$ y $|S_i| = n/p$) y dados $p-1$ pivotes $t_1, t_2, \ldots, t_{p-1}$, tales que $t_1 \leq t_2 \leq \ldots \leq t_{p-1}$ redistribuir la secuencia $S$ en subsecuencias $S'_1, S'_2, \ldots, S'_p$, asignadas a los $p$ procesadores, de manera que todos los elementos resultantes en el procesador $i$, $s'_{ij}$, $j = 1, \ldots, |S'_i|$, sean tales que $s'_{1j} \leq t_1$, $t_{i-1} \leq s'_{ij} \leq t_i$ si $2 \leq i \leq p-1$ y $t_{p-1} \leq s'_{pj}$ (es decir, los elementos estan particionados según los pivotes y todos los elementos de una subsecuencia son menores o iguales a los de la siguiente). Observe que algunas de estas subsecuencias pueden ser vacías.

En primer lugar los procesadores deben determinar donde deben quedar distribuídos sus elementos. Para ello se requiere una difusión (*broadcast*) de los pivotes, y luego una búsqueda del procesador correcto para cada elemento, lo cual puede hacer independientemente

cada procesador en $O(|S_i| \log p) = O(n/p \log p)$ comparaciones usando búsqueda binaria sobre los pivotes. Por último es necesario un intercambio total (*total exchange*) entre los procesadores para redistribuir los elementos nuevos, el cual requiere $p(p-1)$ comunicaciones unidireccionales y $2(p-1)$ pasos de comunicación sobre una topología de conexión completa (ya que sólo se pueden hacer $p/2$ comunicaciones síncronas en paralelo). Si denotamos con *local_distribution*($S_i$) a la partición de la secuencia $S_i$ en un procesador y con *send(x,y)* al envío de los elementos del procesador $x$ que deben ir al $y$, entonces la partición *partition*($S$,$t_1$, ..., $t_{p-1}$) se obtiene con:

*broadcast*($t_1$, ..., $t_{p-1}$)
**for** $i = 1$ **to** $p$ **do in parallel**
    *local_distribution*($S_i$)
**endfor**
**for** $j = 1$ **to** $p - 1$ **do**
    **for** $i = 1$ **to** $p$ **do in parallel**
        **if** $(i/j) \bmod 2 = 0$ **then** *send(i,i + j)*
    **endfor**
    **for** $i = 1$ **to** $p$ **do in parallel**
        **if** $(i/j) \bmod 2 = 1$ **then** *send(i,i + j)*
    **endfor**
**endfor**

Si el número de pivotes es mayor o igual al $p$, el algoritmo puede ser modificado fácilmente asignando uniformemente los pivotes a los procesadores.

Una característica del algoritmo es su falta de estabilidad (la posición relativa de los elementos en $S'_i$ no es necesariamente la misma que en $S$), debido a que el orden final de los elementos en $S'_i$ será el orden de recepción de los mismos durante el intercambio total (si exigimos que sea estable es necesario serializar gran parte de la comunicación).

## 3 *Melsort*

Denotaremos como *melsort(S,n)* al algoritmo secuencial *Melsort* sobre una secuencia $S$ de $n$ elementos y como *melpar(S,n,p)* a la versión paralela propuesta en este trabajo sobre una secuencia $S$ de $n$ elementos usando $p$ procesadores.

Una secuencia de listas anidadas es un conjunto ordenado de listas ordenadas $l_1, \ldots, l_m$ tales que $l_{i+1}$ esta anidada en $l_i$, es decir, el primer elemento de $l_i$ es menor que el primero de $l_{i+1}$ y el último elemento de $l_i$ es mayor que el último de $l_{i+1}$, para todo $i = 1, \ldots, m-1$.

*Melsort* construye un secuencia de listas anidadas y luego procede a su mezcla (*merge*). Durante la construcción de las listas, suponiendo que se han insertado los primeros $j-1$ elementos, el elemento $j$-ésimo es agregado buscando la lista más vieja en la cual pueda ser agregado al principio o al final, preservando el ordenamiento de la lista (para ello se utiliza búsqueda binaria sobre los extremos de las listas); si no puede ser anexado en ninguna de las listas, se crea una nueva lista que lo contenga. El orden del algoritmo es $O(n \log m)$, donde $m$ es la métrica *Enc* aplicada a la secuencia $S$; este orden esta acotado por $O(n \log n)$ y es adaptativo. Sin embargo la optimalidad del algoritmo con respecto a *Enc* es todavía una pregunta abierta.

La primera versión paralela particiona la secuencia de entrada en $p$ subsecuencias aleatorias $S_1, \ldots, S_p$ de igual tamaño $n/p$ y las distribuye entre los procesadores; cada procesador aplica localmente *melsort($S_i$,n/p)*. Al finalizar esta fase se aplica un algoritmo paralelo de mezcla optimal [AS87].

**for** $i = 1$ **to** $p$ **do in parallel**
    *melsort($S_i$,n/p)*
**endfor**
*parallel_merge(S)*

La segunda versión particiona la secuencia de entrada $S$ en $p$ subsecuencias $S'_1, \ldots, S'_p$ de tamaño variable, usando para ello $p-1$ elementos aleatorios, $a_1, \ldots, a_{p-1}$, escogidos como pivotes; sólo resta aplicar localmente *Melsort*.

*partition($S, a_1, \ldots, a_{p-1}$)*

**for** $i = 1$ **to** $p$ **do in parallel**

*melsort*($S'_i$,$n/p$)

**endfor**

# 4 *Slabsort*

*Slabsort* particiona establemente la secuencia original $S$ en $r = \lceil k^2/s \rceil$ subsecuencias $S_1, \ldots, S_r$ de casi el mismo tamaño, donde $k$ es una cota superior para *SMS* de $S$. Para hacer esto se usan los elementos $(n/r + 1)$-ésimo, $(2n/r + 1)$-ésimo, ..., $((r - 1)n/r + 1)$-ésimo como pivotes, lo cual se hace a su vez hallando medianas repetidamente. Para cada intervalo se aplica *melsort*($S_i$,$|S_i|$) con una variante que trata de ordenar la secuencia usando a lo sumo $k$ listas anidadas; si no lo logra la variante retorna un valor que así lo indica. La selección de $r$, asegura que al menos la mitad de estas llamadas son exitosas; en aquellas partes donde *Melsort* no triunfa se aplica recursivamente *Slabsort*. Como inicialmente no se conoce $k$, el algoritmo adivina exponencialmente su valor, es decir, iniciando con un valor de $k = 2$, si en una llamada a *Slabsort* menos de la mitad de las llamadas a *Melsort* son exitosas, entonces *Slabsort* es abortado, el valor de $k$ duplicado y el algoritmo repetido. El costo de este algoritmo es $O(n \log k)$ comparaciones, lo cual es óptimo con respecto a *SMS*.

La versión paralela se divide en dos pasos: en primer lugar hallar las medianas y particionar en subsecuencias; en segundo lugar se procede secuencialmente a aplicar una versión de *melpar*($S_i$,$|S_i|$,$p$) con una cota de $k$ para $m$ en cada sección resultante; en alquellos casos donde no sea exitoso se aplica el procedimiento recursivamente. Para el primer paso se puede usar uno de los algoritmos de selección paralela o la adaptación de uno de ellos para el caso de la mediana [AS87, Akl89].

Alternativamente el último paso puede ser paralelizar el ordenamiento de cada $S_i$ utilizando *melsort*($S_i$,$|S_i|$); dado que $r \neq p$, cada procesador debe ir seleccionando secuencialmente la subsecuencia a ordenar, por lo cual debe existir un administrador de la asignación (en el caso de ser dinámica) o ser prefijada cuando comienza el algoritmo (por ejemplo, de $S_1$ a $S_{r/p}$ para el primer procesador, de $S_{r/p+1}$ a $S_{2r/p}$ para el segundo, etc).

En cualquiera de los casos $k$ es también adivinado exponencialmente.

## 5 *Randomized Slabsort*

*Randomized Slabsort* es similar a *Slabsort* excepto que escoje los pivotes aleatoriamente, particionando establemente en base a ellos. A pesar de que los intervalos son ahora de longitud variable, en el caso promedio el orden del algoritmo sigue siendo $O(n \log k)$, lo cual se conjetura que es óptimo con respecto a *SMS* en el caso promedio.

La versión paralela, similar a la anterior, sólo requiere la particion en subsecuencias para el primer paso. Si suponemos que los datos están inicialmente distribuídos entre todos los procesadores, cada uno de ellos escoje independientemente $r/p$ pivotes aleatorios, procediendo luego a la partición.

## 6 Resultados

El algoritmo paralelo *melpar* fue implantado somo una plataforma de tres transputadores T800 conectados en *pipeline*, cada uno con un *megabyte* de memoria. La versión implantada corresponde a la segunda, donde se aplica el algoritmo de partición señalado usando, en este caso, el número de procesadores menos un pivotes.

Se escogieron varios conjuntos de datos para probar los algoritmos para mostrar las bondades y desventajas del algoritmo: aleatorios, ordenados, invertidos, etc. Por razones de espacio sólo incluímos aquí los resultados obtenidos en el caso aleatorio y en el caso donde la secuencia está semiordenada, constando de tres corridas decrecientes de igual longitud, los cuales se muestran en la figura 1.

En ellas se muestran los tiempos de ejecución de *quicksort* y *melsort* en un tranputador y de *melpar* en tres transputadores, dando el tiempo de ejecución en pulsos de reloj, los cuales son aproximadamente 15000 por segundo. Los algoritmos fueron probados con datos relativamente pequeños debido a las limitaciones de memoria para los algoritmos secuenciales, partiendo con 30000 mil elementos y continuando con incremento iguales hasta 210000.

Como se desprende de los resultados, la eficiencia del algortimo no fue la esperada. Sin embargo todavía falta depurar la implementación y una de las razones por las cuales

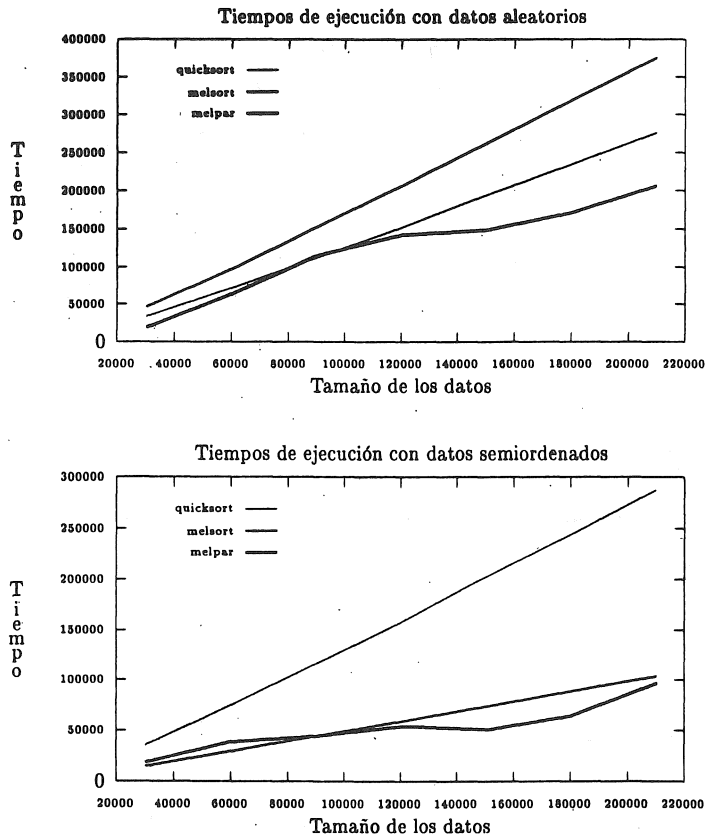


Figura 1: Resultados obtenidos

los resultados no son buenos es la topología de interconexión, la cual convierte a uno de los transputadores en un cuello de botella. La poca diferencia entre el algoritmo paralelo y el secuencial *melsort* puede deberse a la depuración que fue objeto el secuencial, logrando disminuir en 20% el tiempo de ejecución del mismo, mientras que con el paralelo sólo se tiene una versión preliminar.

Otro resultado interesante fue la necesidad de seleccionar los pivotes de manera de minimizar la comunicación, por lo cual se requiere de la implantación de un algoritmo paralelo para búsqueda de medianas.

# Referencias


[Akl89] Selim Akl. *The Design and Analysis of Parallel Algorithms.* Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1989.

[AS87] Selim G. Akl and Nicola Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C–36(11):1367–1369, 1987.

[ECW91] Vladimir Estivill-Castro and Derick Wood. Randomized sorting of shuffled monotone sequences. In *Proceedings of the XI International Conference of the Chilean Computer Science Society*, pages 147–155, 1991.

[LP90] C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. In J.R. Gilbert and R. Karlsson, editors, *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, pages 181–191. Springer–Verlag Lecture Notes in Computer Science 447, 1990.

[Man85] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C–34(4):318–325, 1985.

[Ski88] Steven S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28:775–784, 1988.