Algoritmos Modernos de Ordenación

La librería Boost Sort

CONCEPTOS BÁSICOS I

Ordenación Estable, ordenación no estable

La ordenación **estable** garantiza que 2 ó mas elementos iguales, cuando son ordenados, conservan su **orden relativo**.

230	Antonio Rodriguez

230	Ana Santamaria

Queremos ordenar unos registros por el primer campo.
La ordenación estable garantiza que siempre el registro de Antonio Rodriguez va a ser anterior al de Ana Santamaría

Por lo general, los **algoritmos estables** necesitan **mas memoria** y son **mas lentos** que los algoritmos no estables

CONCEPTOS BÁSICOS II

Datos desordenados y casi ordenados

Cuando pensamos en ordenar pensamos en datos **muy desordenados**. Pero muchas veces no es así. En muchos casos están **“casi” ordenados**. Por ejemplo, datos añadidos al principio o al final

1	3	5	6	7	9	12	14	4	8

O datos insertados ente los elementos ordenados

1	3	5	6	7	15	12	14	4	16

Hay algoritmos muy **rápidos** con **datos desordenados**, pero **lentos** con datos **casi ordenados** (por ejemplo quicksort o stable_sort) y algoritmos **lentos** con datos **desordenados**, pero muy **rápidos** con datos **casi ordenados** (por ejemplo timsort)

CONCEPTOS BÁSICOS III

Datos pequeños, grandes y muy grandes. Strings

En los algoritmos de ordenación, hay 2 tipos de operaciones : **Comparación** y **Movimiento e intercambio** de datos

Con elementos **pequeños** es mucho mas **costosa** la **comparación** que el **movimiento**. Pero con elementos grandes es al revés. Debido a esto hay algoritmos rápidos con elementos pequeños que no lo son con elementos grandes y viceversa.

En las **comparaciones**, podemos encontrar comparaciones **muy sencillas** (por ejemplo comparar por un código que es un número entero), y comparaciones **costosas**, que involucran operaciones complejas (por ejemplo comparar strings).

Por ello hablamos del **coste de la comparación**.

CONCEPTOS BÁSICOS IV

Comparaciones con strings

Los **strings** de C++ son un caso especial, ya que habitualmente son, como poco, un puntero a una memoria que contiene el texto

Esta estructura tiene un **rendimiento pobre de la memoria caché**. Por lo que su **velocidad** puede **variar enormemente**, y algoritmos rápidos con datos grandes y pequeños presentan resultados mediocres con strings, y viceversa.

CONCEPTOS BÁSICOS V

Objeto de comparación. Algoritmos híbridos

Los **algoritmos generales**, para hacer las **comparaciones**, usan un **objeto de comparación**, al que al pasarle los elementos a comparar, nos indica el menor. Estos pueden **ordenar cualquier cosa**, con tal de que tengan su correspondiente objeto de comparación.

En **STL**, por defecto usa **std::less<T>**, que a su vez usa el **operator <** de los objetos a comparar.

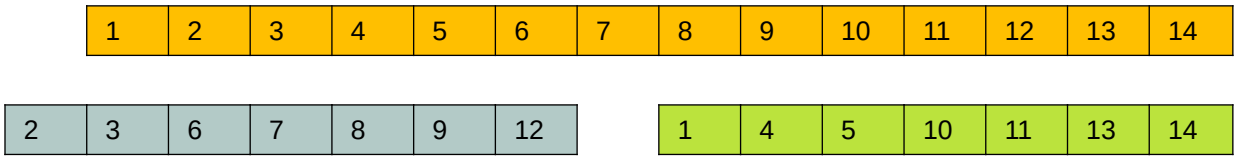
Los **algoritmos híbridos** no utilizan un objeto para comparar, y que **solo ordenan datos de unos determinados tipos (enteros, coma flotante, strings)**. Estos algoritmos suelen ser más rápidos que los que usan objeto de comparación.

Ejemplos de estos son :

- spreadsort
- radixsort
- countingsort
- postmansort
- skasort

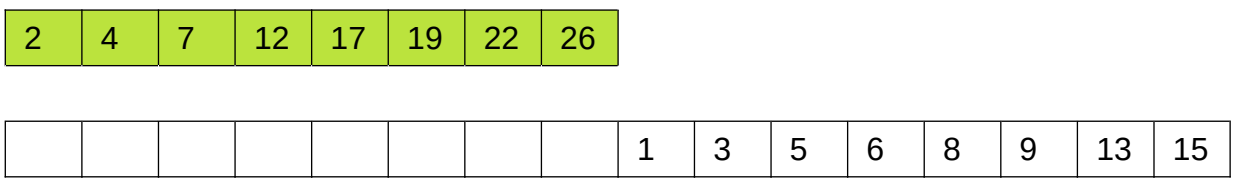
CONCEPTOS BÁSICOS VI

Mezclas de datos



Mezcla simple

La salida ha de tener un tamaño como la suma de las entradas



Media mezcla (half_merge)

La salida ha de tener un tamaño libre como la parte que está en un buffer externo

LA LIBRERIA BOOST SORT

Quienes somos:

- Steven Ross
- Orson Peters
- Francisco Tapia

Esta librería contiene algoritmos de ordenación

- 1 sola hebra (4 algoritmos)
- paralelos (3 algoritmos)

La idea es proporcionar algoritmos que mejoren los proporcionados por compiladores y librerías en el mercado

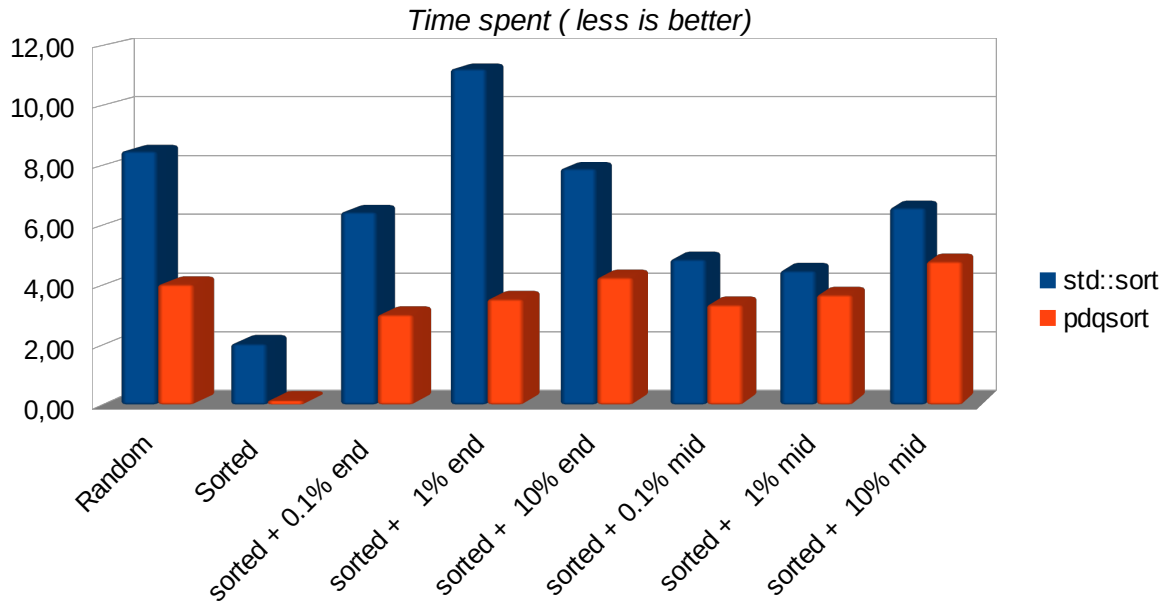
ALGORITMOS DE 1 HEBRA I

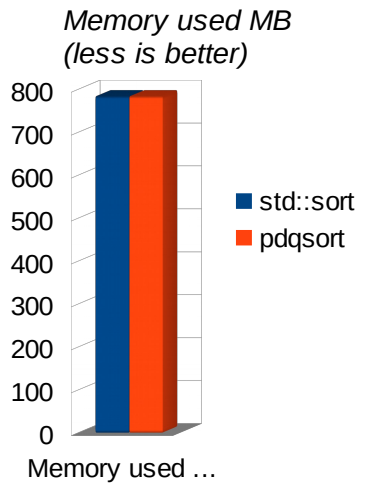
Algorithm	Stable	Additional memory	Best, average, and worst case	Comparison method
spreadsort	no	key_length	N, N sqrt(LogN), min(N logN, N key_length)	Hybrid radix sort
pdqsort	no	Log N	N, N LogN, N LogN	Comparison operator
spinsort	yes	N / 2	N, N LogN, N LogN	Comparison operator
flat_stable_sort	yes	size of the data/256 + 8K	N, N LogN, N LogN	Comparison operator

- **spreadsort** is an extremely fast hybrid radix sort algorithm, designed and developed by Steven Ross.
- **pdqsort** is a improvement of the quick sort algorithm, designed and developed by Orson Peters.
- **spinsort** is a stable sort that is fast with random or nearly sorted data, designed and developed by Francisco Tapia.
- **flat_stable_sort** is a stable sort that uses very little additional memory (around 1% of the size of the data), providing 80% - 90% of the speed of spinsort, designed and developed by Francisco Tapia.

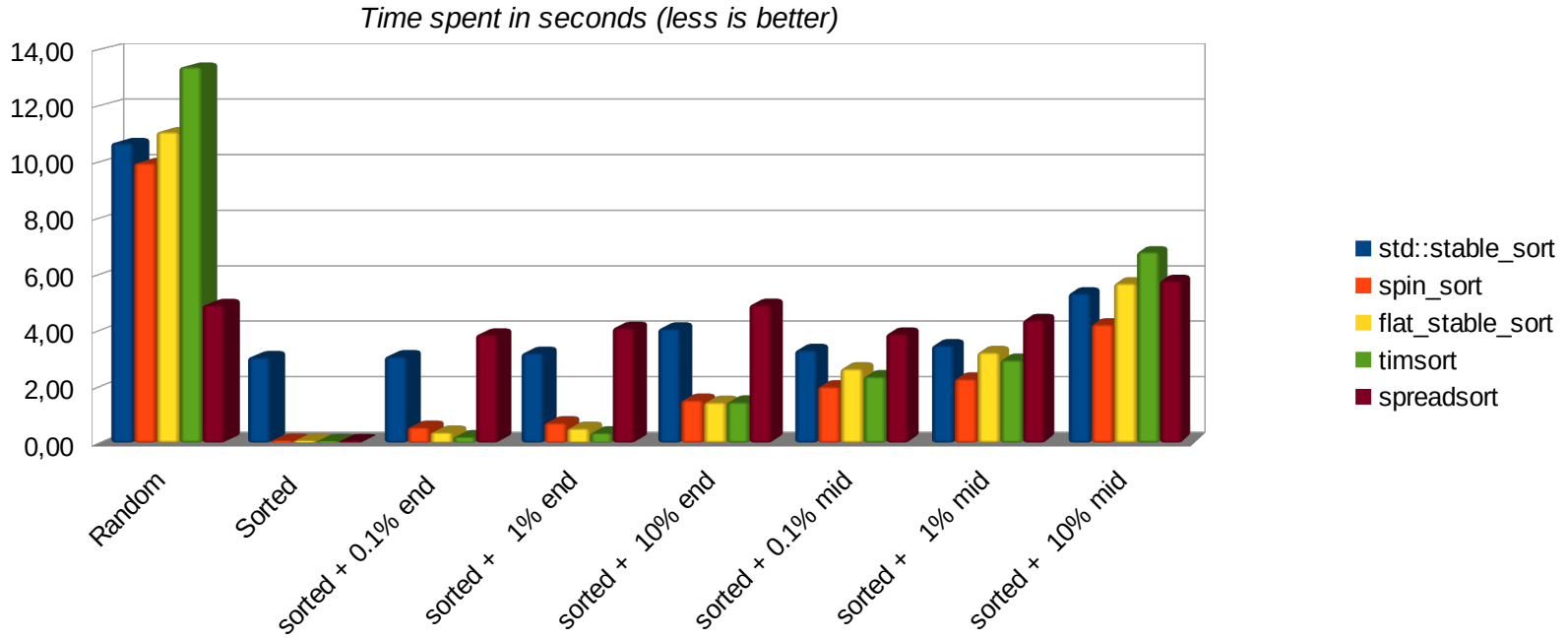
ALGORITMOS DE 1 HEBRA II (no estables)

Este **test** ordena **100 000 000 número de 64 bits**, tanto completamente **desordenados** , como **casi ordenados**, añadiendo elementos desordenados al final , o insertándolos en el medio



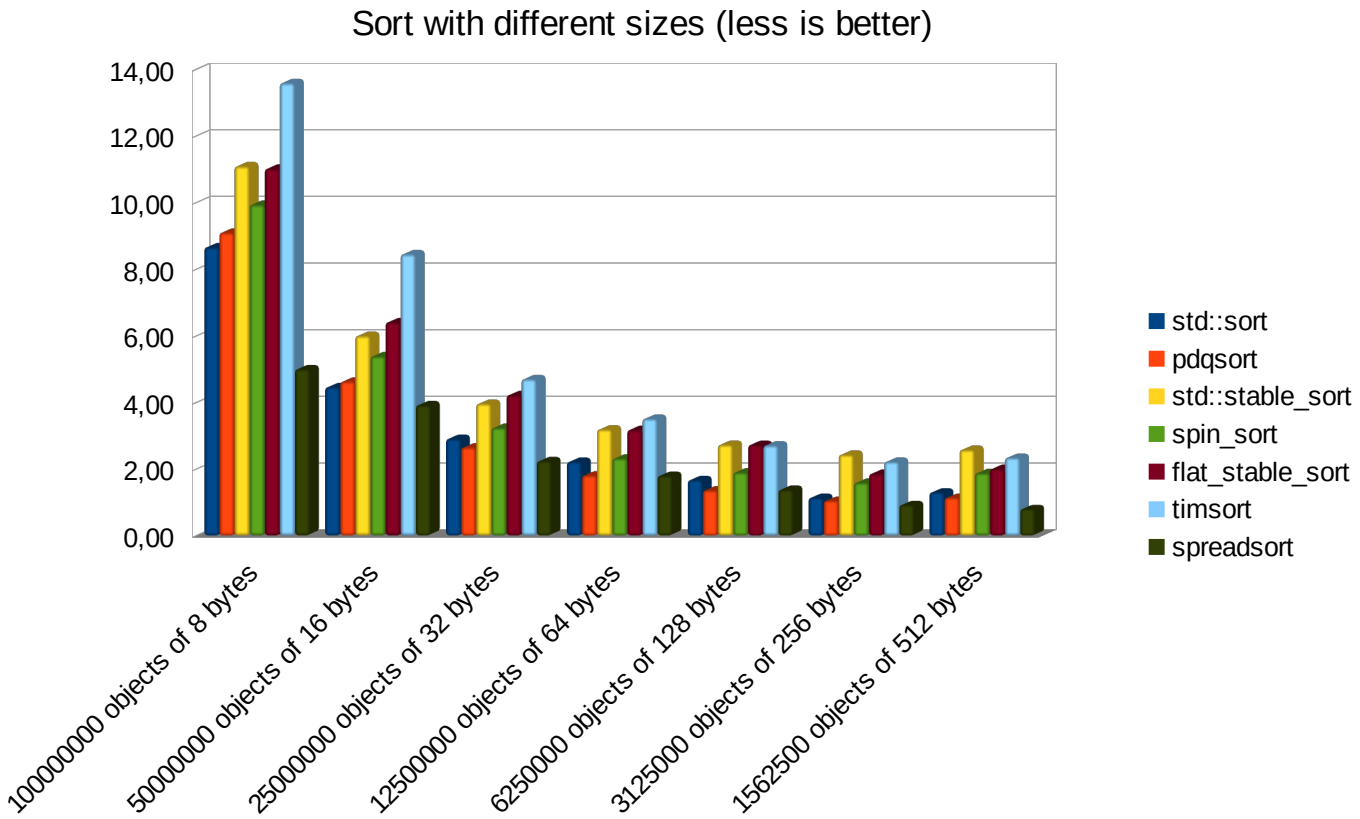


ALGORITMOS DE 1 HEBRA III (estables)



Las **ordenaciones** con elementos **casi ordenados**, son mucho mas **frecuentes** de lo que nos imaginamos en el **mundo real**

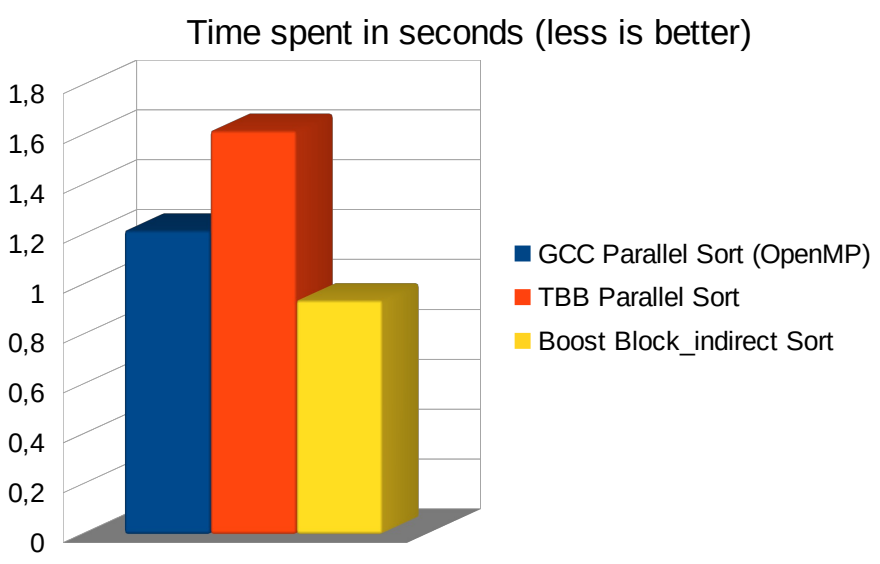
ALGORITMOS DE 1 HEBRA IV (varios tamaños)

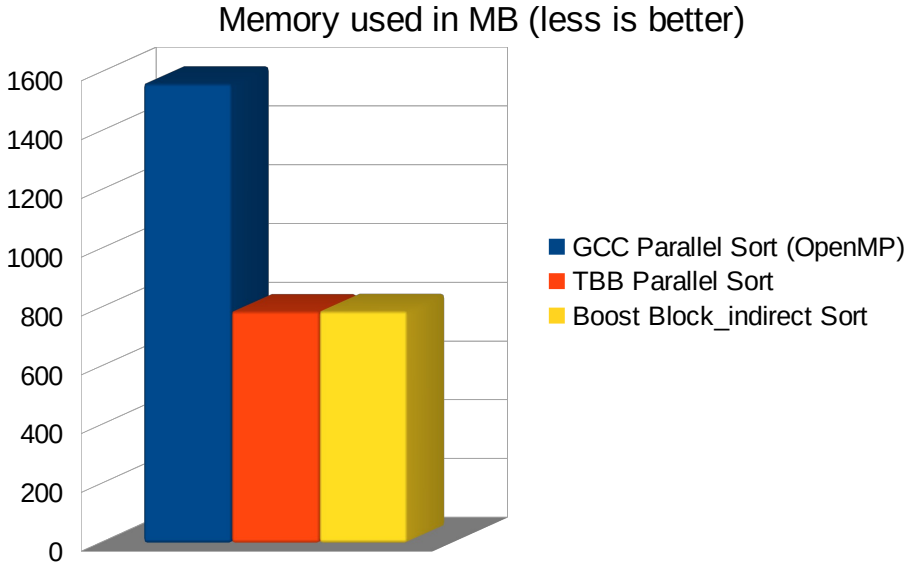


En la ordenación **estable**, el mas **lento** es siempre **timsort**, y el mas **rápido** es **spin_sort.**

Es meritoria la **velocidad** de **flat_stable_sort** , que **no usa memoria adicional.**

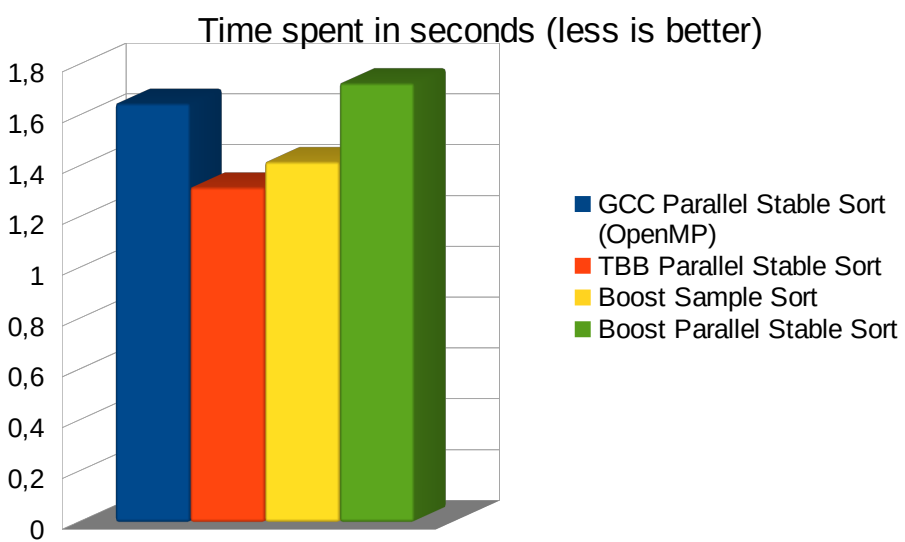
LINUX ALGORITMOS PARALELOS I (no estables)

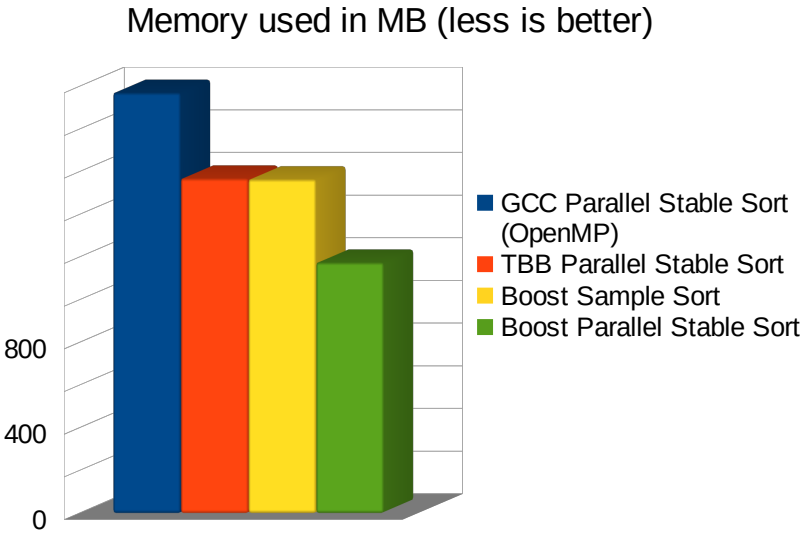




La librería **Boost** introduce un **nuevo algoritmo paralelo** (**block_indirect_sort**), ideado e implementado por Francisco Tapia, que combina una **gran velocidad** con un **bajo consumo de memoria**, usando una **novedosa técnica** que se describirá mas adelante.

LINUX ALGORITMOS PARALELOS II (estables)

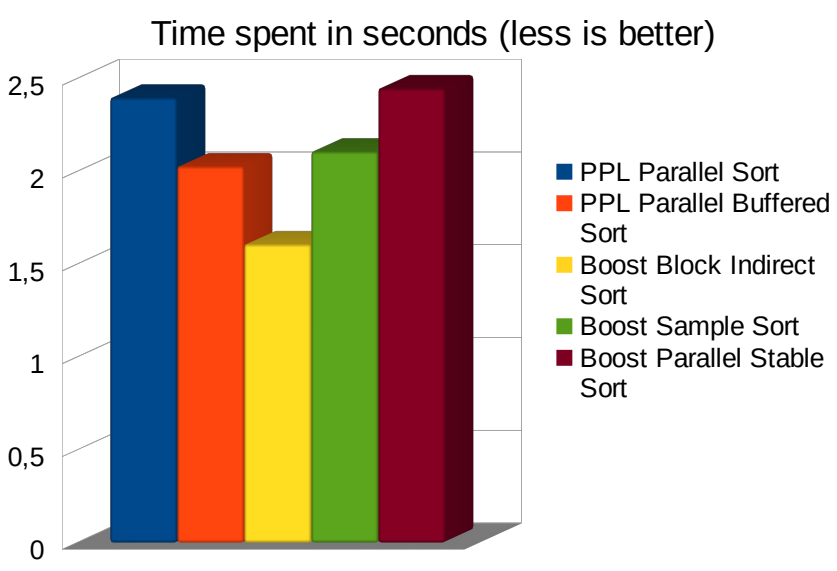


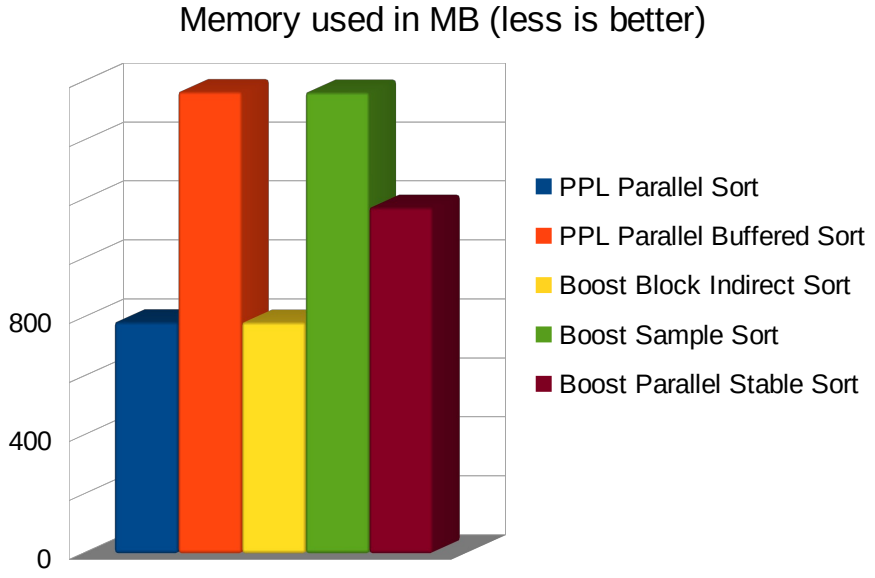


Los algoritmos **TBB Parallel Stable Sort** y **Boost Sample Sort**, son implementan el algoritmo **sample_sort. Boost Parallel Stable Sort** internamente utiliza SampleSort, y trata de reducir la memoria adicional que usa a la mitad.

GCC Paralle Stable Sort utiliza un **algoritmo antiguo** y una implementación que **no reusa la memoria adicional,** por lo que es **lento** y consume **mucha memoria.**

WINDOWS ALGORITMOS PARALELOS I



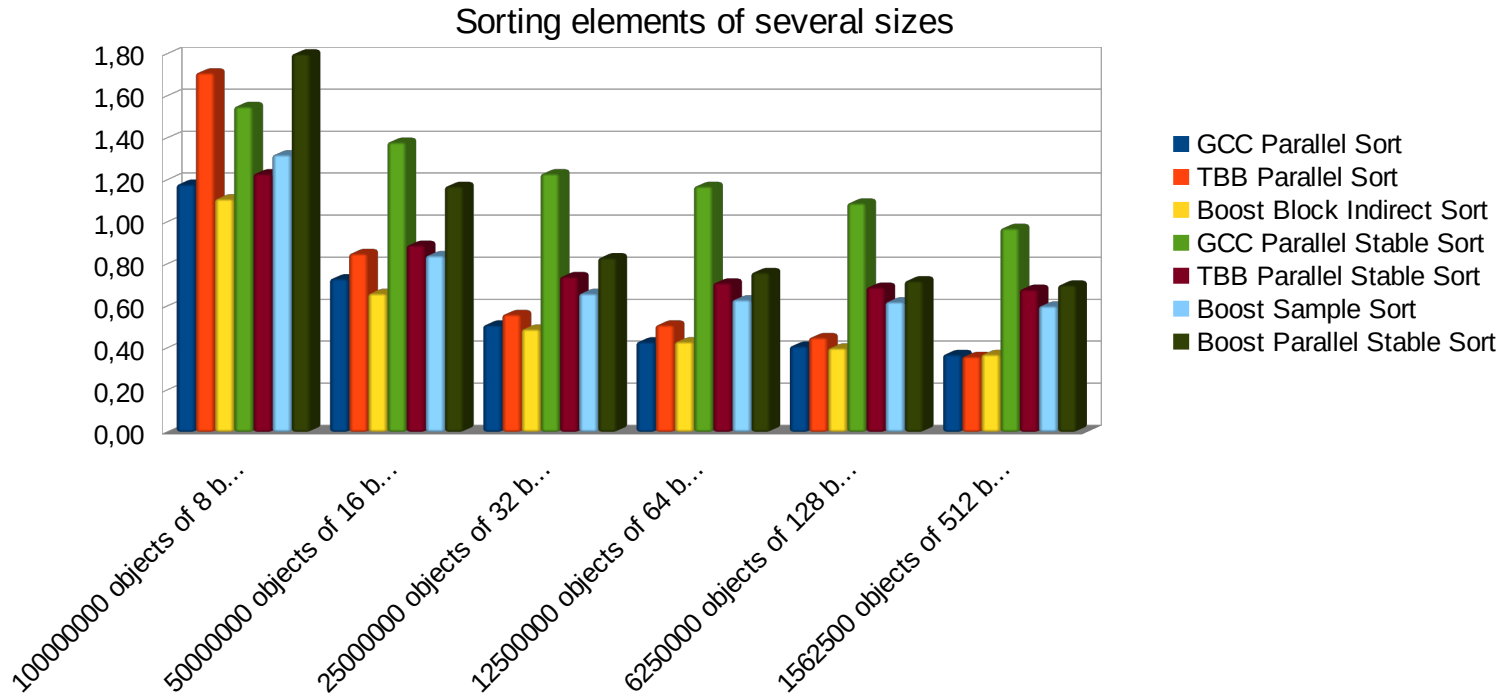


Microsoft tiene 2 algoritmos paralelos diferentes.

- **PPL Parallel Sort** es **similar** a un **QuickSort paralelo**, consume **poca memoria**, pero es **lento**.
- **PPL Parallel Buffered Sort** es un algoritmo **parecido** a **GCC Parallel Sort**. Es mas **rápido**, pero consume el **doble de memoria**.

Microsoft **no tiene** algoritmos **estables paralelos.**

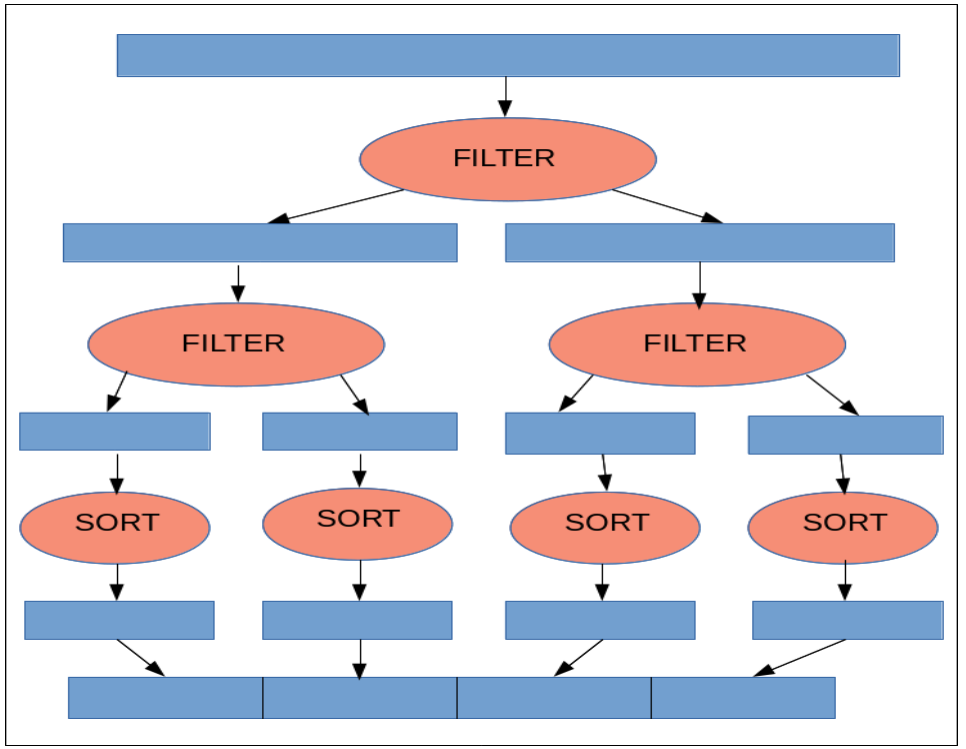
ALGORITMOS PARALELOS (Diferentes Tamaño)



Con los **tamaños grandes,** los **tiempos son similares** porque el **bus de datos** está **saturado**.

Los **no estables** hacen **menos movimientos** de los datos para ordenar, por eso, cuando el **tamaño de los datos crece**, la **diferencia** es tan **acusada**.

QUICKSORT PARALELO

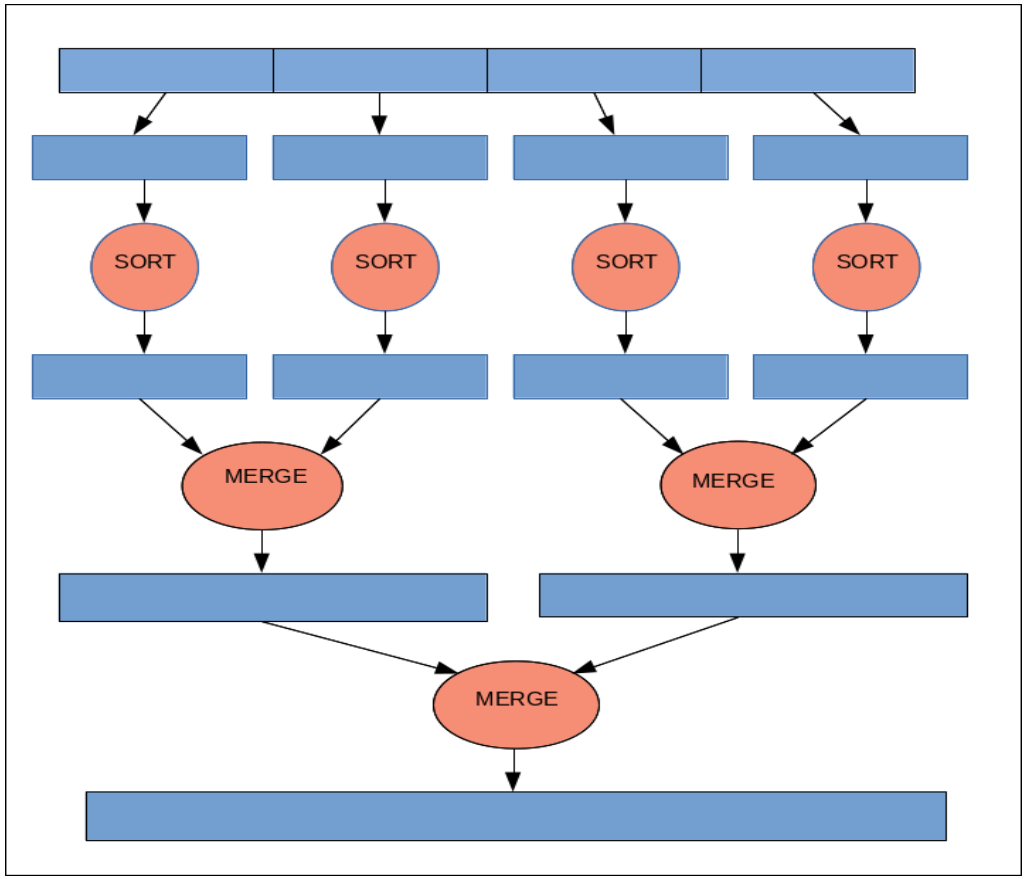


Una **hebra** hace un **filtrado por un valor**, a un lado los mayores y a otro los menores.

Cada **parte** a su vez es **dividida** por otra hebra, hasta que la subdivisión produce suficientes partes para tener a todas las hebras ocupadas.

El algoritmo es **rápido** y no necesita **memoria adicional**, pero los **tiempos no son buenos cuando el número de hebras crece**

PARALLEL STABLE SORT



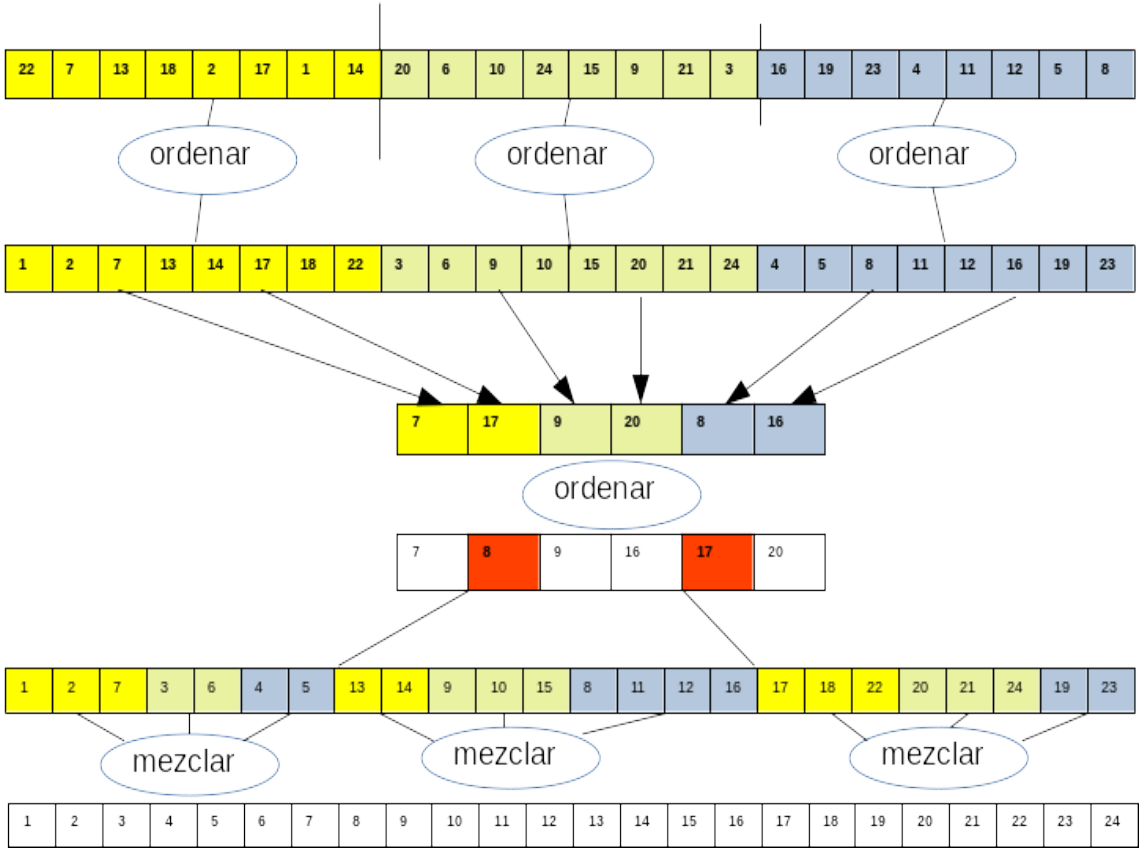
Inicialmente **divide** los datos en muchas partes. Cada **parte** es **ordenada** por un **hebra**.

Luego se van **mezclando** hasta que solo queda **una parte ,** en la que los elementos están ordenados.

Estos algoritmos **necesitan memoria adicional** para hacer la mezclas, normalmente del **mismo tamaño que los datos**

Tiene el mismo **problema** que **parallel quick sort**. La **parte final se hace con una sola hebra**. Por eso no es rápido cunado el número de hebras crece

SAMPLE SORT



1.- **Dividimos** en partes iguales , que **ordenamos**

2.- De cada parte **sacamos** unos **pivotes** uniformemente distanciados.

3.- **Ordenamos los pivotes** de todas las partes, y sacamos **otros pivotes**, uniformemente distanciados

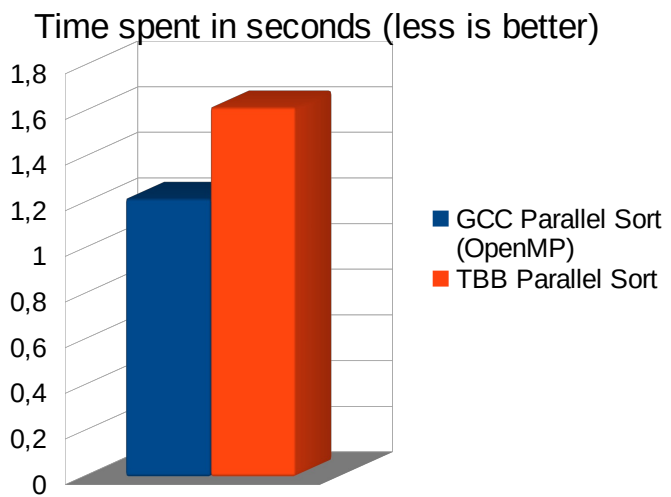
4.- Copiamos los **elementos menores** que el primer pivote de la cada parte en una memoria auxiliar, con lo que obtenemos un **conjunto de rangos a mezclar**

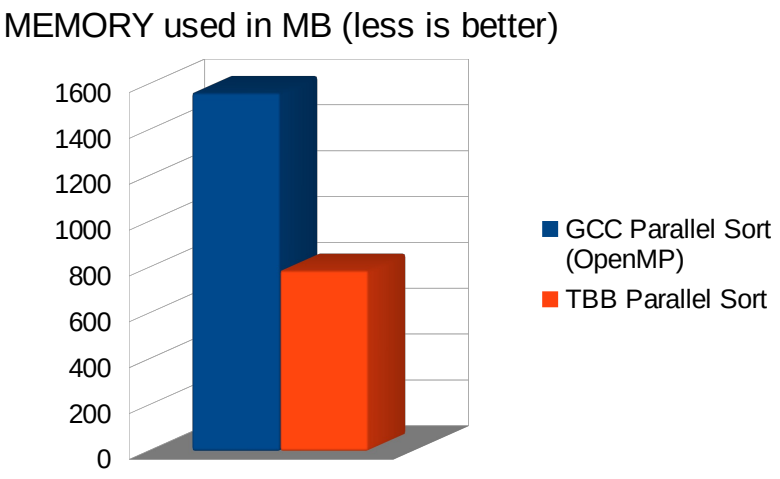
5.- Hacemos lo mismo con el segundo pivote, y así sucesivamente

6.- Se **mezclan los rangos** obtenidos de cada pivote y los datos ya están ordenados

CREANDO UN NUEVO ALGORITMO PARALELO

Queremos construir un **algoritmo rápido** como GCC Parallel Sort, pero que utilice **poca memoria** como TBB Parallel Sort

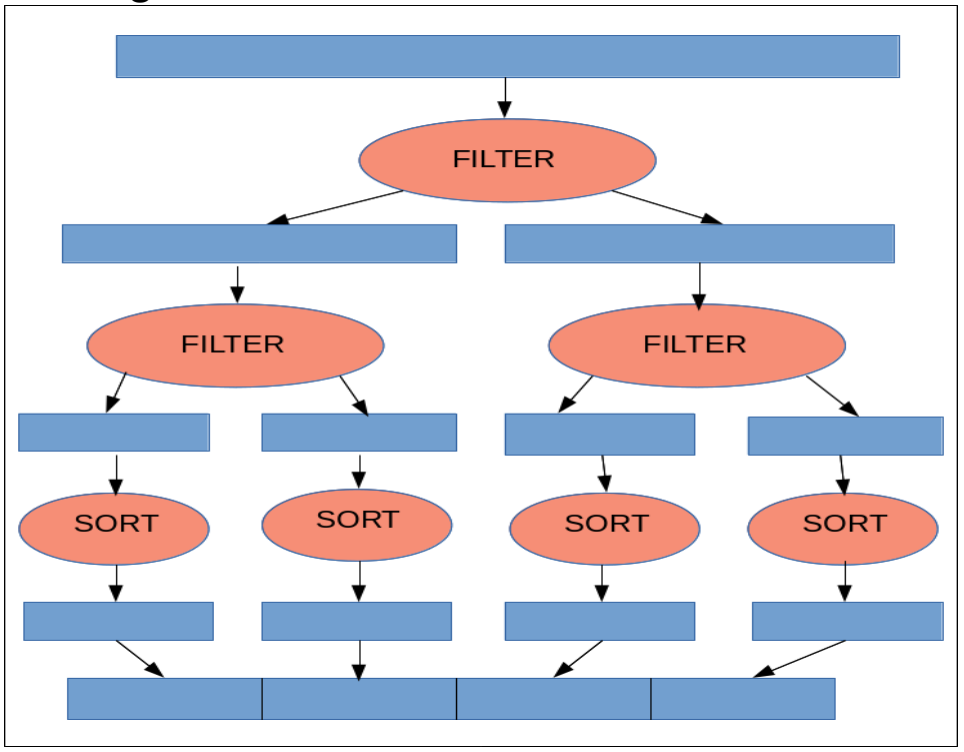




DISEÑO I (Que sabemos)

Los algoritmos de ordenación paralelos son de dos grandes tipos

1.- **Algoritmos de subdivisión o Filtrado**. Como quicksort paralelo o TBB Parallel Sort.



Una **hebra** hace un **filtrado** por un **valor**, a un lado los mayores y a otro los menores.

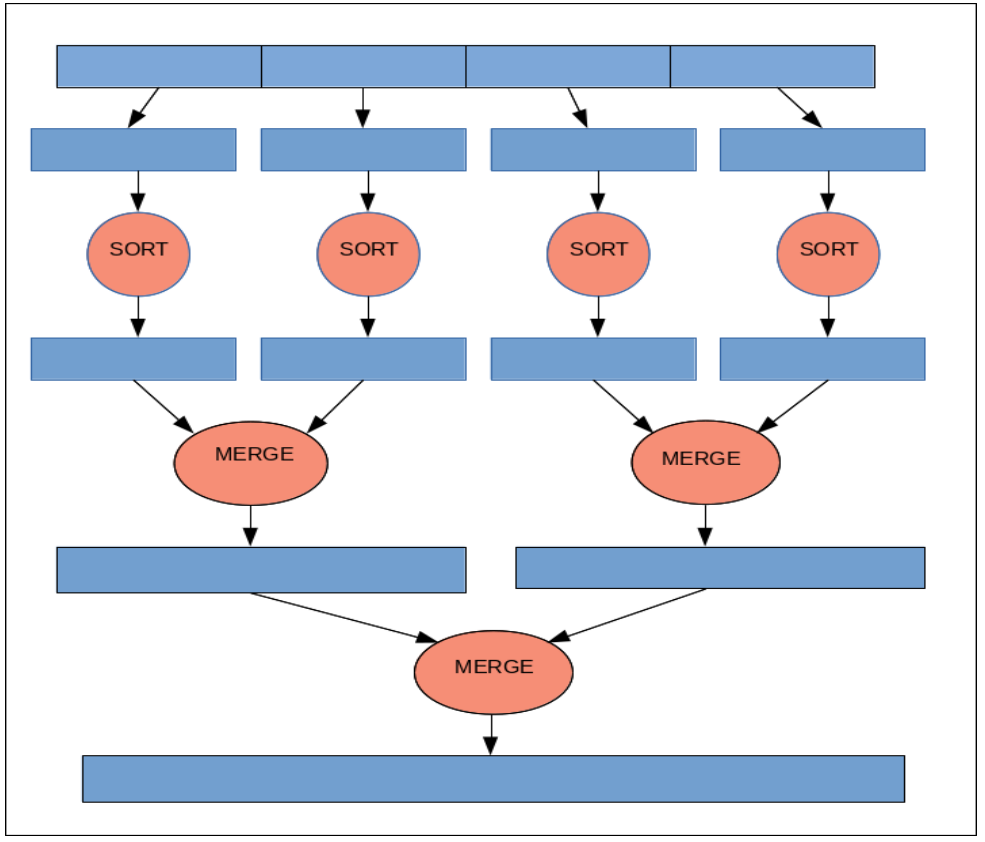
Cada **parte** a su vez es **dividida** por otra hebra, hasta que la subdivisión produce suficientes partes para tener a todas las hebras ocupadas.

El algoritmo es rápido y no necesita memoria adicional, pero los tiempos no son buenos cuando el número de hebras crece.

Si no somos **capaces** de hacer el **filtrado paralelo**, este camino no tiene salida

DISEÑO II (Que sabemos)

2.- Algoritmos de Mezcla. Como GCC Parallel Sort, SampleSort



Inicialmente **divide los datos** en muchas partes. Cada parte es **ordenada por un hebra**.

Cuando las **partes** están **ordenadas**, se van **mezclando** hasta que solo queda una parte , en la que los elementos están ordenados.

Estos algoritmos **necesitan memoria adicional** para hacer la mezclas, normalmente del **mismo tamaño que los datos**

Estos algoritmos proporcionan las mejores prestaciones con **muchas hebras**, pero con pocas hebras su resultado es inferior a los algoritmos de filtrado.

Las **mezclas** han de ser **paralelizables** para que el algoritmo sea **rápido**

DISEÑO III (Que buscamos)

“Hacer lo mismo que otros, nos conduce, después de mucho trabajo, a parecidos resultados”

- El **algoritmo que queremos hacer** ha de ser de **mezcla**, que **genera trabajo** para todas las hebras del procesador, porque los **algoritmos de filtrado** tienen **mal rendimiento** cuando el **número de hebras crece.**
- El **consumo de memoria** ha de ser lo mas **bajo** posible.
- Para que sea **rápido**, todas las **partes del algoritmo** han de ser **paralelizables**

Pero ...¿ **como hacer la mezcla sin**, o con una pequeña **memoria adicional** ?

La respuesta es **considerar** que los **elementos** están en **bloques de tamaño fijo**, excepto el último o bloque cola. Para **mezclar dos bloques de longitud fija**, solo **necesitamos** otro **bloque** auxiliar del mismo tamaño. Ese **bloque auxiliar** va a ser nuestra única memoria adicional.

DISEÑO IV (Mezcla de bloques)

Inicialmente tenemos **2 bloques** con **datos ordenados** y un **bloque vacío**

Los vamos **mezclando sobre el bloque vacío**, hasta llenarlo

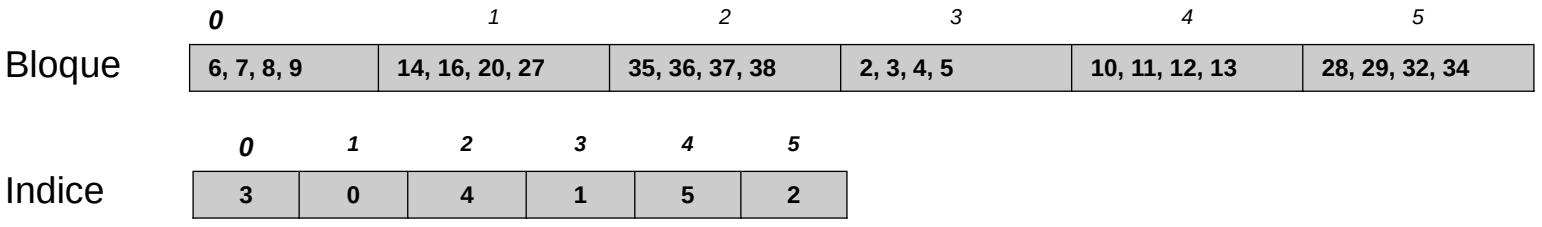
Hacemos un **half_merge** de un bloque semivacío sobre el otro, y obtenemos un **bloque vacío** y uno lleno

DISEÑO V (Ordenación física y lógica)

Cuando **mezclamos** los **bloques**, apuntamos su **orden relativo** en un **índice**.

Por esto decimos que los **bloques** no están físicamente ordenados, sino que están **lógicamente ordenados mediante un índice**

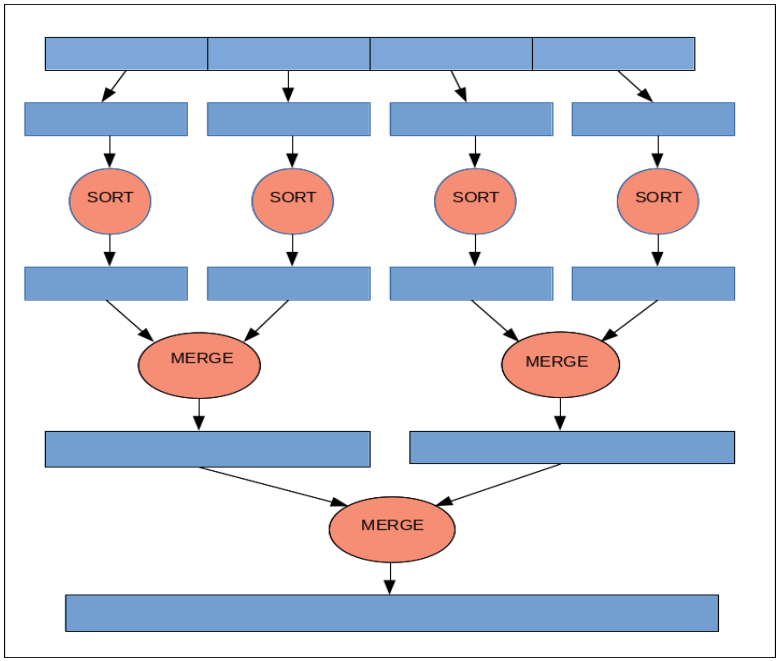
Debido a esto, **cada bloque** va a tener una **posición física** dentro del vector de bloques, y una **posición lógica** que nos va a indicar el orden de dichos bloques.



DISEÑO VI (Mezclando los bloques)

Part 1		Part 2		First merge		Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Final Merge
				2 5 9 10		2 3 4 5					2 3 4 5
2 5 9 10		3 4 6 7		3 4 6 7		6 7 9 10	6 7 8 9				6 7 8 9
12 28 32 34		8 11 13 14		8 11 13 14			10 11 13 14	10 11 12 13			10 11 12 13
35 37 39 40		16 20 27 29		12 28 32 34				14 28 32 34	14 16 20 27		14 16 20 27
				16 20 27 29					28 29 32 34		28 29 32 34
				35 37 39 40						35 36 37 38	35 36 37 38

DISEÑO VII (¿Como partir la mezcla?)



En los **primeros pasos** tenemos **trabajo para todas las hebras**, pero conforme avanzamos el número de hebras que trabajan se divide por dos, y el **último paso** se hace con **una sola hebra**.

Es el **mismo problema** que presentan los **algoritmos de filtrado**.

¿Y si **esas mezclas pudiéramos trocearlas** para que se pudieran **ejecutar en paralelo**?

DISEÑO VIII (¿Como partir la mezcla?)

Para poder **partir la mezcla**, necesitamos **saber** a priori el **orden** exacto de los **bloques a mezclar.** Si nos fijamos en el **dibujo** de la **página anterior**, veremos que están **ordenados** por el **primer valor del bloque**.

Para **partir** esta **lista** en **dos partes** que se puedan ejecutar en paralelo.

- Para ello buscamos **dos bloques contiguos** en dicha lista que sean de **diferente color.**
- **Mezclamos** esos **dos bloques** solamente. El **bloque** resultante con los **datos mas pequeños**, pertenece a la **parte superior**, y el **otro bloque**, pertenece a la **parte inferior.**
- Dichas **partes** se pueden **mezclar en paralelo** por diferentes hebras, proporcionando dicho proceso una **solución correcta**.

Cualquier **mezcla** se puede **dividir** en **cualquier numero de partes**, que pueden ser ejecutadas en paralelo.

DISEÑO IX (Partiendo en dos la mezcla)

Part 1		Part 2		First merge		Mezcla para partir la lista	Pass 1	Pass 2	Final Merge
				2 5 9 10			2 3 4 5		2 3 4 5
2 5 9 10		3 4 6 7		3 4 6 7			6 7 9 10	6 7 8 9	6 7 8 9
12 28 32 34		8 11 13 14		8 11 13 14		8 11 12 13		10 11 12 13	10 11 12 13
35 37 39 40		16 20 27 29		12 28 32 34		14 28 32 34	14 16 20 27		14 16 20 27
				16 20 27 29			28 29 32 34	28 29 32 34	28 29 32 34
				35 37 39 40				35 36 37 38	35 36 37 38

La **división** la hacemos entre el **tercer** y **cuarto bloque** de la columna first merge, que son de diferente color.

Los **mezclamos** y el **bloque superior** pertenece a la **partición de arriba** y el **bloque inferior** a la **partición de abajo**.

Obtenemos **dos listas independientes** que se pueden **mezclar por separado**.

Las **fusiones grandes** e van a **descomponer** en **múltiples fusiones** para ser ejecutadas en paralelo

DISEÑO X (Reordenación de Bloques)

D

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
200 201	500 549	600 654	900 980	100 127	400 453	700 739	800 861

I

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
4	0	5	1	2	6	7	3

Después de las fusiones, tenemos todos los datos ordenados por su posición lógica mediante el índice.

Ahora queremos moverlos para que estén ordenados también por su posición física

```
pos_dest = pos_ini = 0
pos_src = I [pos_dest]
Aux = D [pos_dest]
do
{   D [pos_dest] = D [pos_src]
    I [pos_dest] = pos_dest
    pos_dest = pos_src
    pos_src = I [pos_dest]
} while (pos_src != pos_ini )
D [pos_dest] = Aux
I [pos_dest] = pos_dest
```

Si apuntamos lo que hemos hecho, aparece:

Aux ← bloque[0]
bloque[0] ← bloque[4]
bloque[4] ← bloque[2]
bloque[2] ← bloque[5]
bloque[5] ← bloque[6]
bloque[6] ← bloque[7]
bloque[7] ← bloque[3]
bloque[3] ← bloque[1]
bloque[1] ← Aux

Podríamos representarlo como una **secuencia cerrada o ciclo** : 0, 4, 2, 5, 6, 7, 3, 1

DISEÑO XI (Reordenación de bloques)

En las reordenaciones de bloques **pueden aparecer varios ciclos.** Habitualmente aparecen unos pocos ciclos grandes y varios pequeños.

En el siguiente ejemplo

Datos

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
24	23	20	15	12	10	21	17	19	13	22	18	14	11	16

Indice

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	13	4	9	12	3	14	7	11	8	2	6	10	1	0

Haciendo el procedimiento descrito anteriormente, en este caso nos aparecen 3 secuencias

- 5, 3, 9, 8, 11, 6, 14, 0
- 4, 12, 10, 2
- 13, 1

DISEÑO XII (Reordenación de bloques)

El **peor caso** es cuando aparece **una sola secuencia**. Si ese caso **aparece echa al traste todo el algoritmo**, ya que al tener que hacerlo con **una hebra**, los **tiempos se disparan**.

La solución es **trocear las secuencias**, y generar **varias secuencias** mas **pequeñas** que se puedan ejecutar en **paralelo**

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
Datos	100	140	70	60	90	00	160	80	50	130	150	20	170	10	110	30	120	40

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
Indice	5	13	11	15	17	8	3	2	7	4	0	14	16	9	1	10	6	12

Si extraemos la secuencia como hemos visto anteriormente tenemos

5	8	7	2	11	14	1	13	9	4	17	12	16	6	3	15	10	0

DISEÑO XIII (Reordenación de bloques)

Si la secuencia la **dividimos en 3 partes**, obtenemos 3 secuencias de 6 elementos

5	8	7	2	11	14	1	13	9	4	17	12	16	6	3	15	10	0

Antes de mover cada secuencia, **generamos** una secuencia con las últimas posiciones de cada una de ellas.

14	12	0

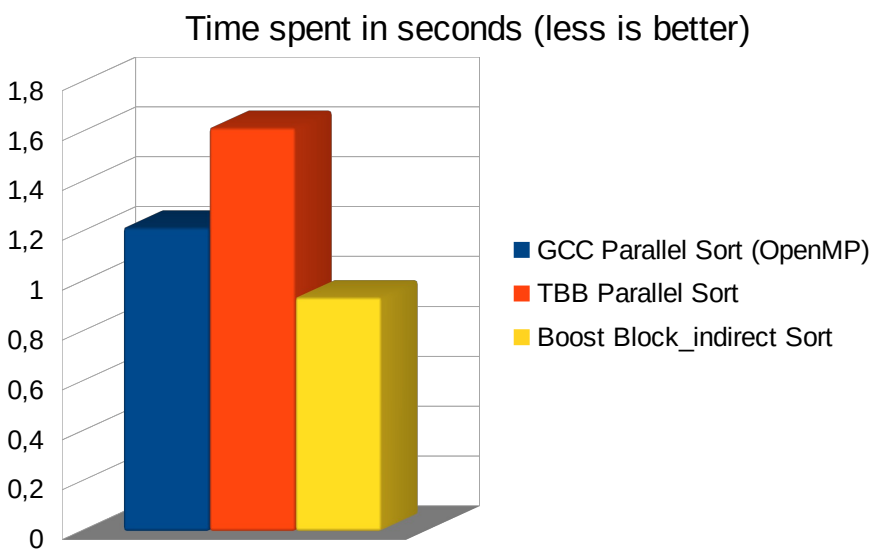
Cada uno de los **3 movimientos** de las secuencias se puede **ejecutar en paralelo**. Y al acabar los 3 movimientos, **movemos** la secuencia con los **últimos elementos**. Y ya está la secuencia grande movida.

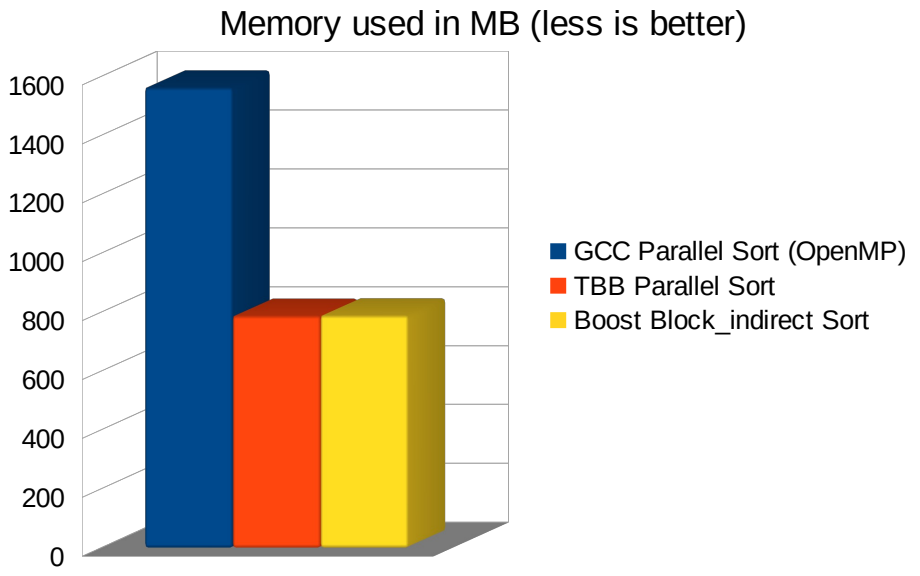
Y hemos **troceado** una **secuencia grande** en un número arbitrario de **secuencias pequeñas**, que se pueden **ejecutar en paralelo**. **EL ALGORITMO YA ESTÁ COMPLETAMENTE DISEÑADO**.

Ahora solo queda la **codificación**, las **pruebas** y una **cuidadosa optimización**. La hora de la verdad son las pruebas de **rendimiento**

DISEÑO XIV (Rendimiento)

Los resultados obtenidos son :





Tenemos un **algoritmo** con un **consumo muy bajo de memoria** y una **gran velocidad**. Todos sus procesos internos son **altamente paralelizables**, lo que hace este algoritmo especialmente **eficiente** en entornos con **muchas hebras**.

¿Y DESPUÉS QUE ? I

La evolución lógica pasa por dos proyectos

TERASORT- Ordenación en memoria compartida con muchas máquinas.

Los **algoritmos actuales** usan un **40% - 45 %** de la **memoria** de las máquinas usadas. Por lo que si tenemos que ordenar 1 Tera en memoria, y tenemos máquinas con 128 Gigas de RAM, necesitaremos unas 20 máquinas.

He **diseñado un algoritmo en papel**, que permite usar un **95% de la memoria** de la máquina. Por lo que en el caso anterior se podría hacer con 9 ó 10 máquinas de 128G de RAM.

Lo he comentado con varios centros de Supercomputación y Big Data y les he preguntado si lo ven útil. Todos me dijeron que **no es un problema**, que **basta con poner mas máquinas**. Por lo cual, me parece que el algoritmo no saldrá de mi libreta

¿Y DESPUÉS QUE? II

ALGORITMO ESTABLE PARALELO SIN MEMORIA ADICIONAL . Algo **parecido a Block_indirect_sort** pero con **ordenación estable**

Su diseño fue un reto que me puse a mi mismo. Pero he preguntado a bastante gente, y el interés mostrado ha sido muy bajo, por lo que lo he puesto junto al algoritmo de Terasort para que se hagan compañía.

BIBLIOGRAFIA

Introduction to Algorithms, 3rd Edition (Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein)

Structured Parallel Programming: Patterns for Efficient Computation (Michael McCool, James Reinders, Arch Robison)

Algorithms + Data Structures = Programs (Niklaus Wirth)

Boost Sort Library
www.boost.org/doc/libs/1_69_0/libs/sort/doc/html/index.html

Detailed description of the block_indirect_sort algorithm
www.boost.org/doc/libs/1_69_0/libs/sort/doc/papers/block_indirect_sort_en.pdf

Detailed description of the flat_stable_sort algorithm
www.boost.org/doc/libs/1_69_0/libs/sort/doc/papers/flat_stable_sort_eng.pdf