# BLOCK INDIRECT

## *A new parallel sorting algorithm*

*Francisco Jose Tapia*
*fjtapia@gmail.com*

## BRIEF

Modern processors obtain their power increasing the number of “cores” or HW threads, which permit them to execute several processes simultaneously, with a shared memory structure.

## SPEED OR LOW MEMORY

In the parallel sorting algorithms, we can find two categories .

**SUBDIVISION ALGORITHMS**
Filter the data and generate two or more parts. Each part obtained is filtered and divided by other threads, until the size of the data to sort is smaller than a predefined size, then it is sorted by a single thread. The algorithm most frequently used in the filter and sort is quick sort.

These algorithms are fast with a small number of threads, but inefficient with a large number of HW threads. Examples of this category are :

- ◦Intel Threading Building Blocks (TBB)
- ◦Microsoft PPL Parallel Sort.

**MERGING ALGORITHMS**
Divide the data into many parts at the beginning, and sort each part with a separate thread. When the parts are sorted, merge them to obtain the final result. These algorithms need additional memory for the merge, usually an amount equal to the size of the input data.

With a small number of threads, these algorithms usually have similar speed to the subdivision algorithms, but with many threads they are much faster . Examples of this category are :

- ◦GCC Parallel Sort (based on OpenMP)
- ◦Microsoft PPL Parallel Buffered Sort

## SPEED AND LOW MEMORY

This new algorithm is an unstable parallel sort algorithm, created for processors connected with shared memory. This provides excellent performance in machines with many HW threads, similar to the GCC Parallel Sort, and better than TBB, with the additional advantage of lower memory consumption.

This algorithm uses as auxiliary memory a 1024 element buffer for each thread. The worst case memory usage for the algorithm is when elements are large and there are many threads. With big elements (512 bytes), and 32 threads, the memory measured was:

- GCC Parallel Sort 1565 M
- Threading Building Blocks (TBB) 783 M
- Block Indirect Sort 814 M

# *INDEX*

# *1.- OVERVIEW OF THE PARALLEL SORTING ALGORITHMS*

Among the unstable parallel sorting algorithms, there are basically two types:

### 1.- SUBDIVISION ALGORITHMS

As Parallel Quick Sort. One thread divides the problem in two parts. Each part obtained is divided by other threads, until the subdivision generates sufficient parts to keep all the threads buys. The below example shows that this means with a 32 HW threads processor, with N elements to sort.

| Step | Threads working | Threads waiting | Elements to process by each thread |
|---|---|---|---|
| 1 | 1 | 31 | N |
| 2 | 2 | 30 | N / 2 |
| 3 | 4 | 28 | N / 4 |
| 4 | 8 | 24 | N / 8 |
| 5 | 16 | 16 | N / 16 |
| 6 | 32 | 0 | N / 32 |

Very even splitting would be unusual in reality, where most subdivisions are uneven

This algorithm is very fast and don't need additional memory, but the performance is not good when the number of threads grows. In the table before, until the 6th division, don't have work for to have busy all the HW threads, with the additional problem that the first division is the hardest, because the number of elements is very large.

### 2.- MERGING ALGORITHMS,

Divide the data into many parts at the beginning, and sort each part with a separate thread. When the parts are sorted, merge them to obtain the final result. These algorithms need additional memory for the merge, usually an amount equal to the size of the input data.

These algorithms provide the best performance with many threads, but their performance with a low number of threads is worse than the subdivision algorithms.

# *2.- INTERNAL DESCRIPTION OF THE ALGORITHM*

This new algorithm (Block Indirect), is a merging algorithm. It has similar performance to GCC Parallel Sort With many threads, but using a low amount of additional memory, close to that used by subdivision algorithms.

This new algorithm only need an auxiliary memory of 1024 elements for each HW thread. The worst case memory use is with big elements and many HW threads. The measured memory with objects of 512 bytes, with a 32 HW threads is:

| ALGORITHM USED | Memory (M Bytes) |
|---|---|
| TBB | 1565 |
| Parallel Sort (GCC) | 3131 |
| Block Indirect | 1590 |

These algorithms are not optimal when using just 1 thread, but are easily parallelized, and need only a small amount of auxiliary memory.

The algorithm divide the number of elements in a number of parts that is the first power of two greater than or equal to the number of threads to use. ( For example: with 3 threads, make 4 parts, for 9 threads make 16 parts…) Each part obtained is sorted by the parallel intro sort algorithm.

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td></tr>
<tr><td colspan="2">1-2</td><td colspan="2">3-4</td><td colspan="2">5-6</td><td colspan="2">7-8</td><td colspan="2">9-10</td><td colspan="2">11-12</td><td colspan="2">13-14</td><td colspan="2">15-16</td></tr>
<tr><td colspan="4">1-2-3-4</td><td colspan="4">5-6-7-8</td><td colspan="4">9-10-11-12</td><td colspan="4">13-14-15-16</td></tr>
<tr><td colspan="8">1-2-3-4-5-6-7-8</td><td colspan="8">9-10-11-12-13-14-15-16</td></tr>
<tr><td colspan="16">1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16</td></tr>
</table>

With the sorted parts, merge pairs of parts.

We consider the elements are in blocks of fixed size. We initially restrict the number of elements to sort (N) to a multiple of the block size. For to explain the algorithm, I use several examples with a block size of 4. Each thread receives a number of blocks to sort, and when complete we have a succession of blocks sorted.

For the merge, we have two successions of block sorted. We sort the blocks of the two parts the first element of the block. This merge is not fully sorted, is sorted only by the first element. We call this First Merge

But if we merge the first block with the second, the second with the third and this successively, we will obtain a new list fully sorted with a size of the sum of the blocks of the first part plus the blocks of the second part. This merge algorithm, only need an auxiliary memory of the size of a block.

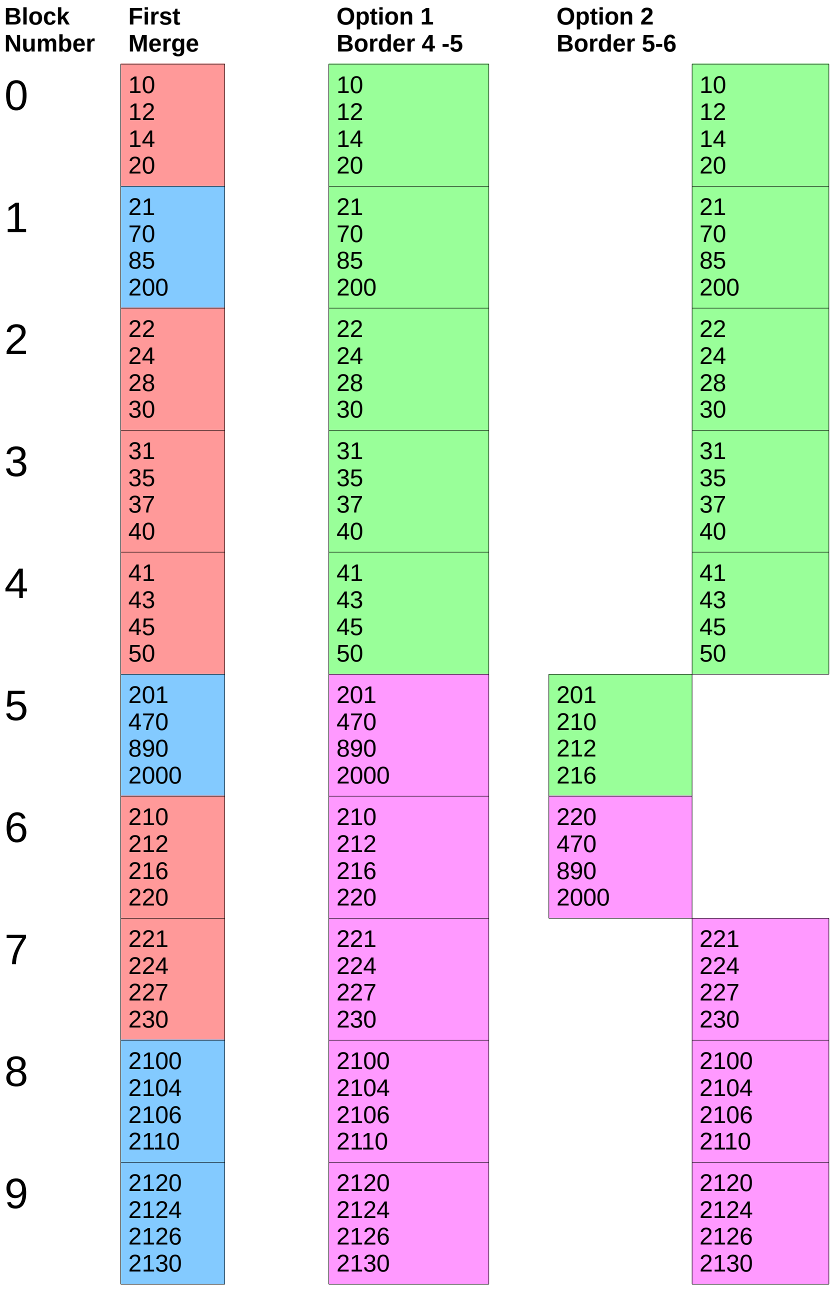
| **Part 1** | **Part 2** | **First merge** | **Pass 1** | **Pass 2** | **Pass 3** | **Pass 4** | **Final Merge** |
|---|---|---|---|---|---|---|---|
| | | 2<br>5<br>9<br>10 | 2<br>3<br>4<br>5 | | | | 2<br>3<br>4<br>5 |
| 2<br>5<br>9<br>10 | 3<br>4<br>6<br>7 | 3<br>4<br>6<br>7 | 6<br>7<br>9<br>10 | 6<br>7<br>8<br>9 | | | 6<br>7<br>8<br>9 |
| 12<br>28<br>32<br>34 | 8<br>11<br>13<br>14 | 8<br>11<br>13<br>14 | | 10<br>11<br>13<br>14 | 10<br>11<br>12<br>13 | | 10<br>11<br>12<br>13 |
| 35<br>37<br>39<br>40 | 16<br>20<br>27<br>29 | 12<br>28<br>32<br>34 | | | 14<br>28<br>32<br>34 | 14<br>16<br>20<br>27 | 14<br>16<br>20<br>27 |
| 44<br>46<br>50<br>71 | 36<br>38<br>45<br>60 | 16<br>20<br>27<br>29 | | | | 28<br>29<br>32<br>34 | 28<br>29<br>32<br>34 |
| | | 35<br>37<br>39<br>40 | 35<br>36<br>37<br>38 | | | | 35<br>36<br>37<br>38 |
| | | 36<br>38<br>45<br>60 | 39<br>40<br>45<br>60 | 39<br>40<br>44<br>45 | | | 39<br>40<br>44<br>45 |
| | | 44<br>46<br>50<br>71 | | 46<br>50<br>60<br>71 | | | 46<br>50<br>60<br>71 |

The idea which make interesting this algorithm, is that you can divide, in an easy way, the total number of blocks to merge,  in several parts, obtaining several groups of blocks, independents between them, which can be merged in parallel.

## 2.1.- MERGE SUBDIVISION

Suppose, we have the next first merge, with block size of 4, and want to divide in two independent parts, to be executed by different threads.

To divide, we must looking for a border between two blocks of different color. And make the merge between them. In the example, to obtain parts of similar size , we can cut in the frontier 4-5, or in the border 5 -6, being the two options.



| Block Number | First Merge | Option 1 Border 4 -5 | Option 2 Border 5-6 |
|---|---|---|---|
| 0 | 10<br>12<br>14<br>20 | 10<br>12<br>14<br>20 | 10<br>12<br>14<br>20 |
| 1 | 21<br>70<br>85<br>200 | 21<br>70<br>85<br>200 | 21<br>70<br>85<br>200 |
| 2 | 22<br>24<br>28<br>30 | 22<br>24<br>28<br>30 | 22<br>24<br>28<br>30 |
| 3 | 31<br>35<br>37<br>40 | 31<br>35<br>37<br>40 | 31<br>35<br>37<br>40 |
| 4 | 41<br>43<br>45<br>50 | 41<br>43<br>45<br>50 | 41<br>43<br>45<br>50 |
| 5 | 201<br>470<br>890<br>2000 | 201<br>470<br>890<br>2000 | 201<br>210<br>212<br>216 |
| 6 | 210<br>212<br>216<br>220 | 210<br>212<br>216<br>220 | 220<br>470<br>890<br>2000 |
| 7 | 221<br>224<br>227<br>230 | 221<br>224<br>227<br>230 | 221<br>224<br>227<br>230 |
| 8 | 2100<br>2104<br>2106<br>2110 | 2100<br>2104<br>2106<br>2110 | 2100<br>2104<br>2106<br>2110 |
| 9 | 2120<br>2124<br>2126<br>2130 | 2120<br>2124<br>2126<br>2130 | 2120<br>2124<br>2126<br>2130 |

In the option 1, the frontier is between the blocks 4 and 5, and the last of the block 4 is less or equal than the first of the block 5, and don't need merge. We have two parts, which can be merged in a parallel way The first with the blocks 0 to 4, and the second with the 5 to 9.

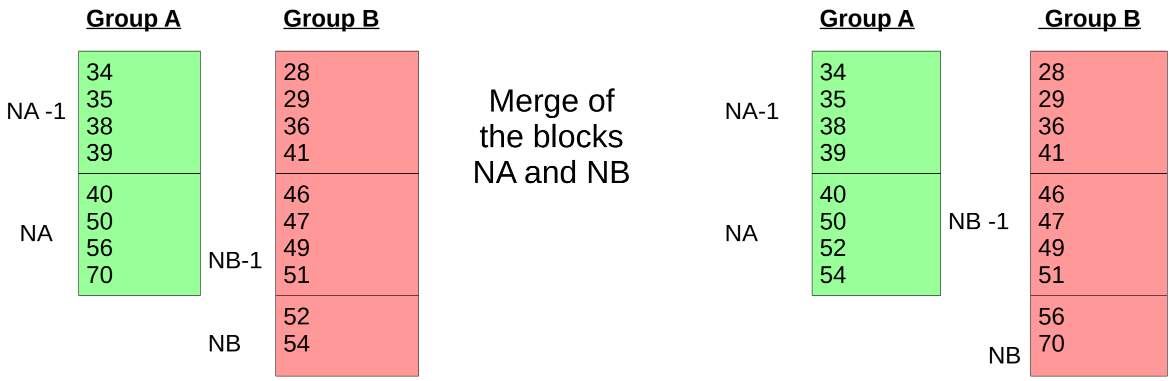
In the option 2, the frontier is between the blocks 5 and 6, and the last of the block 5 is greater than the first of the block 6, and we must do the merge of the two blocks, and appear new values for the blocks 5 and 6. Then we have two parts, which can be merged in a parallel way. The first with the blocks from 0 to 5, and the second with the 6 to 9.

## 2.2.- NUMBER OF ELEMENTS NOT MULTIPLE OF THE BLOCK SIZE

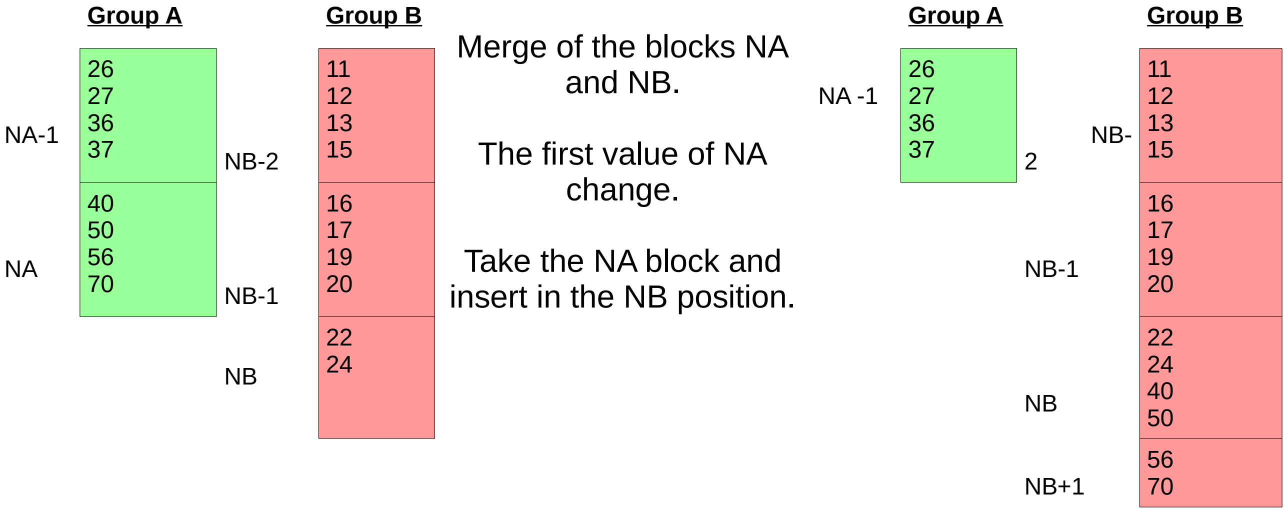
Until now, we assumed the number of elements is a multiple of the block size. When this is not true, we configure the blocks, beginning for the position 0, and at end we have an incomplete block called tail. The tail block always is in the last part to merge. We use a special operation just for this block, described in the next example :

We have two groups of blocks A and B, with NA and NB blocks respectively. The tail block, if exist, always is in the group B. Merge the tail block with the last block of the group A. With the merge two cases appear shown in the next examples:

**Case 1:** The first value of the NA block don't change, and don't do nothing with the blocks.



**Case 2:** The first value of the block A, changes, and we delete the block of the group A and insert in the group B, immediately before the tail block. With this operation we guarantee the tail block is the last



## 2.3.- INDIRECT SORT

In today's computers, the main bottleneck is the data bus. When the process needs to manage memory in memory different locations, as in the parallel sorting algorithms, the data bus limits the speed of the algorithm.

In the benchmarks done on a 32 HW threads, sorting N elements of 64 Bytes of size requires the same time as sorting N / 2 elements 128 bytes of size. The comparison is the same with the two sizes.

In the algorithm described, in each merge, the blocks are moved, to be merged in the next step. This slows down the algorithm, due to the data bus bottleneck.

To avoid this, do an indirect merge. We have an index with the relative position of the blocks. This implies the blocks to merge are not contiguous, but that doesn't change the validity of the algorithm.

When all the merges are complete, with this index we move the blocks, and have all the data sorted. This block movement is done with a simple and parallel algorithm,

## 2.4.- IN PLACE REARRANGEMENT FROM A INDEX (BLOCK SORTING)

The tail block, if one exists, is always in the last position, and doesn't need to be moved. We move blocks with the same size using an index. The best way is to see with an example :

We have an unsorted data vector (D) with numbers. We also have, an index ( I ), which is a vector with the relative position of the elements of D

*D*

| *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* |
|---|---|---|---|---|---|---|---|
| 200 | 500 | 600 | 900 | 100 | 400 | 700 | 800 |

*I*

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

To move the data, we need an auxiliary variable (Aux) of the same size as the data. In this example the data is a number, but in the case of blocks, the variable must have the size of a block.

This can be a bit confusing. Here are the steps of the process :

- Aux = D[0], and after this we must find which element must be copied in the position D[0], we find it in the position 0 of the index, and it is the position 4.
- The next step is D[0] = D[4], and find the position to move to the position 4, in the position 4 of the index, and this is the position 2.
- When doing this successively, once the new position obtained is the first position used, (in this example is the 0), move to this position from the Aux variable, and the cycle is closed.

In this example the steps are:

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

If we follow the arrows, we see a closed cycle. This cycle has a sequence formed by the position of the elements passed. In this example the sequence is 0, 4, 2, 5, 6, 7, 3, 1.

With small elements the sequence is useless, because instead of extracting the sequence, we can move the data, and all is done. But with big elements, as the blocks used in this algorithm, the sequence are very useful, because from the sequences, we generate the parallel work for the threads.

There can be several cycles In an index, with their corresponding sequences. To extract the sequences, begin with the index.

- If in a position, the content is the position, indicate this element is sorted, and don't need to be moved.

- If it's different, this imply it's the beginning of a cycle, and must extract the sequence, as described before, and determine the positions visited in the index.

We can see with an example of an index with several cycles.

**Data vector (D)**

| *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 |

**Index (I)**

| *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 |

Doing the procedure previously described,  find 3 sequences

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

In the real problems, usually appear a few long sequences, and many small sequences. This permits parallel execution, but it's not very efficient, because a long sequence can keep one thread busy while the other threads are waiting, because they are finished with the sort sequences. Or even worse, there can be just one sequence.

To deal with this issue, the long sequences can be easily divided and done in parallel. This permit an optimal parallelization.

## 2.5- SEQUENCE PARALLELIZATION

The procedure can be a bit confusing, so here’s an example:

**Data vector (D)**

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

**Index vector (I)**

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

If we extract the sequence, as described before, we find only one loop, and one sequence. This sequence is;

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

We want to divide in 3 sequences of 6 elements. Each sequence obtained are  independent between them and can be done in parallel. The procedure is :

Generate a fourth sequence with the contents on the last position of each sequence. In this example is

| | | |
|---|---|---|
| 14 | 12 | 0 |

Now, consider the 3 sequences as independent and can be applied in parallel. We apply the sequences over the vector of data (D), and when all are finished, apply the sequence obtained with the last position of the sub sequences. And all is done

See this example:

The data vector is

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

Apply this sequence

| | | | | | |
|---|---|---|---|---|---|
| 5 | 8 | 7 | 2 | 11 | 14 |

The new data vector is

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

Apply this sequence

| | | | | | |
|---|---|---|---|---|---|
| 1 | 13 | 9 | 4 | 17 | 12 |

The new data vector is

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

Apply this sequence

| | | | | | |
|---|---|---|---|---|---|
| 16 | 6 | 3 | 15 | 10 | 0 |

The new data vector is

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

These 3 sub sequences can be done in parallel, because don't have any shared element.
Finally, when the subsequences are been applied, we apply the last sequence, obtained with the last positions of the sub sequences

| | | |
|---|---|---|
| 14 | 12 | 0 |

The new data vector is fully sorted

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

# 3.- *BENCHMARKS*

## 3.1.- INTRODUCTION

To benchmark this we use  the implementation proposed for the Boost Sort Parallel Library. It's pending of the final approval, due this can suffer some changes until the final version and definitive approval in the boost library. You can find in https://github.com/fjtapia/sort_parallel.

If you want run the benchmarks in your machine, you can find the code, instructions and procedures in https://github.com/fjtapia/sort_parallel_benchmark

For the comparison, we use these parallel algorithms:

1. GCC Parallel Sort
2. Intel TBB Parallel Sort
3. Block Indirect Sort

## 3.2.- DESCRIPTION

This benchmark was run on a Dell Power Edge R520 12 G 2.1 GHz with two
Intel Xeon E5-2690, with the Hyper Threading activate and with 32 HW threads.

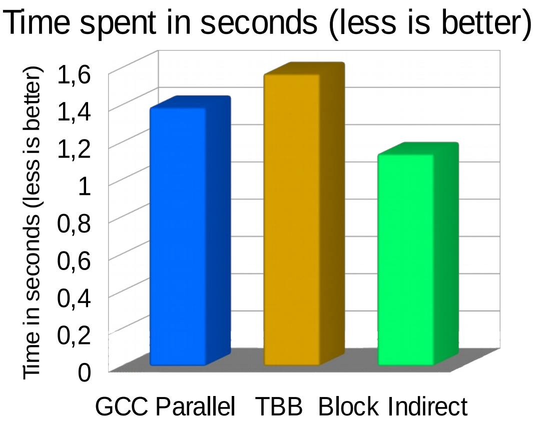
The compiler used was the GCC 5.2  64 bits

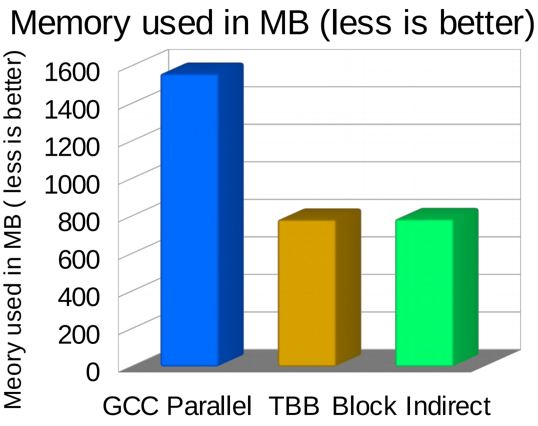
The benchmark have 3 parts:

- Sorting of 100 000 000 64 bits numbers
- Sorting of 10 000 000 strings
- Sorting of objects of different sizes, with different comparison methods.

## 3.3.- NUMBERS

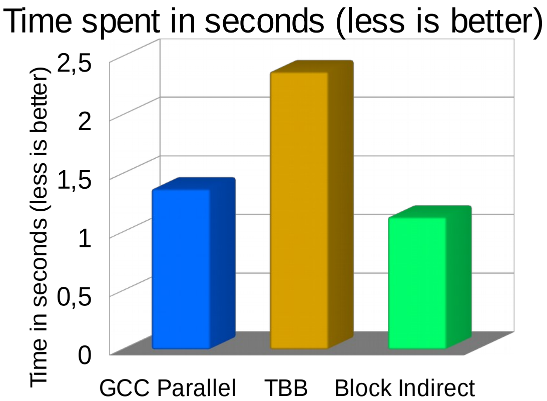
Sorting of 100 000 000 random 64 bits numbers.

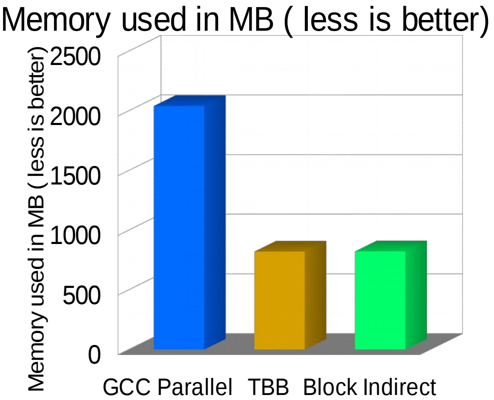




## 3.4.- STRINGS

Sorting 10 000 000 strings, randomly filled



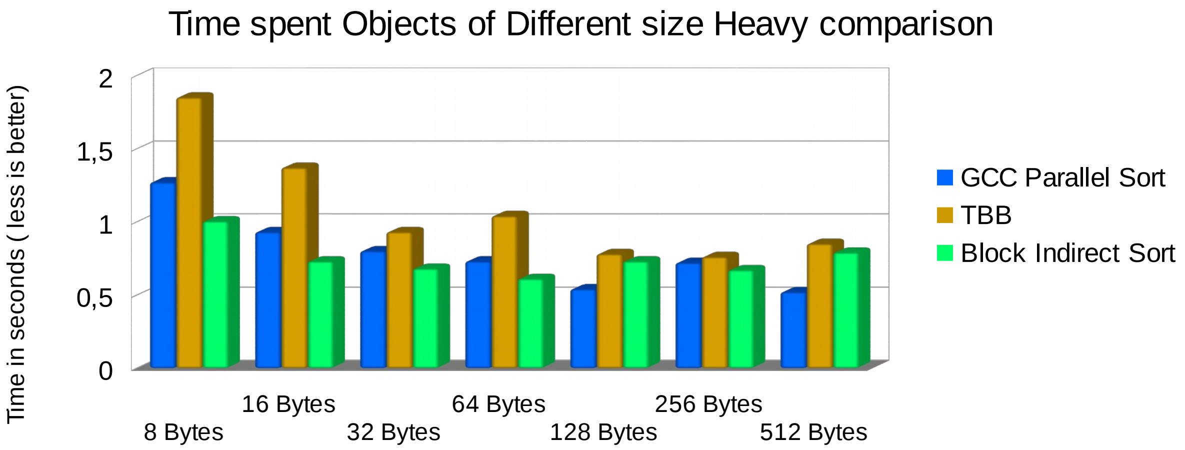


## 3.5.- OBJECTS

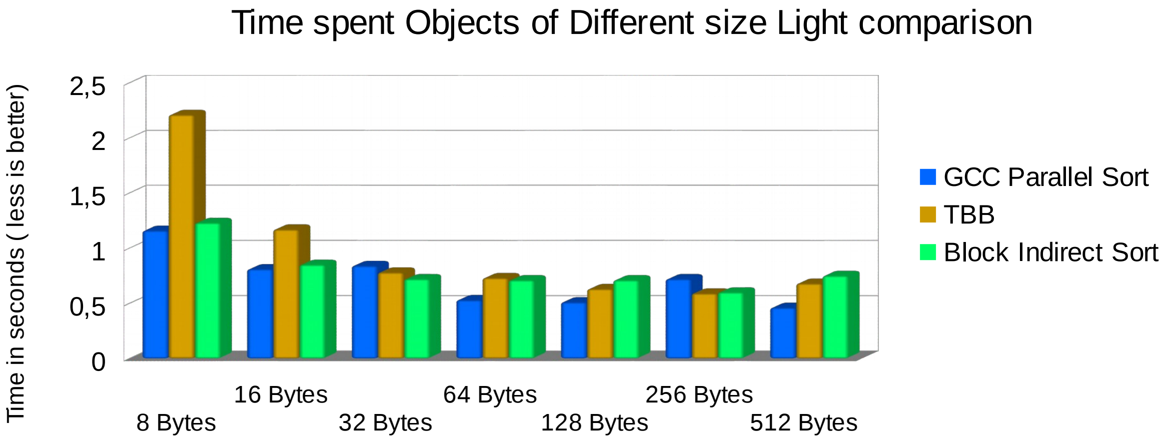
The objects are arrays of 64 bits numbers, randomly filled. An 8 byte object is an array of 1 element, a 16 bytes object have 2 numbers and successively , until the object of 512 bytes, which have 64 numbers.

The benchmark had been done using two kind of comparison:

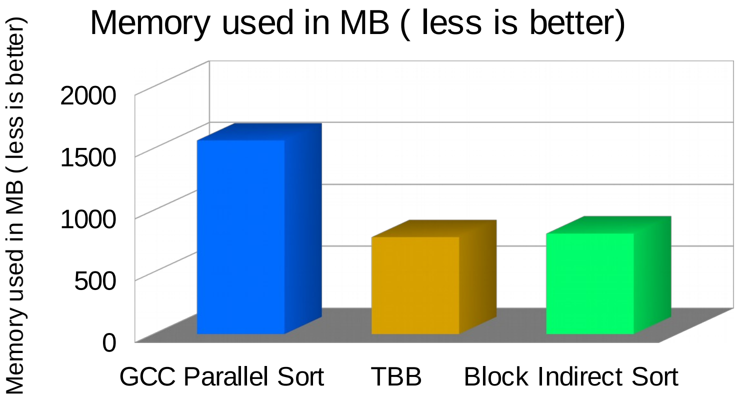
**1.- Heavy comparison**. The comparison is the sum of all the numbers in the array. In each comparison, the sum is done



**2.- Light comparison.** The comparison is done comparing the first element of the array as a key.



### 3.- Memory used



# 4.- BIBLIOGRAPHY

- **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)

# 5.- GRATITUDE

To **CESVIMA** (http://www.cesvima.upm.es/), **Centro de Cálculo de la Universidad Politécnica de Madrid.** When need machines for to tune this algorithm, I contacted with the investigation department of many Universities of Madrid. Only them, help me.

To **Hartmut Kaiser**, Adjunct Professor of Computer Science at Louisiana State University. By their faith in my work,

To **Steven Ross**, by their infinite patience in the long way in the develop of this algorithm, and their wise advises.