BLOCK INDIRECT

A new parallel sorting algorithm

** Francisco Jose Tapia*
fjtapia@gmail.com

BRIEF

The modern processors obtain their power increasing the number of “cores” or HW threads, which permit to execute several process 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 thread, 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 sorting is quick sort.

These algorithms are fast with a small number of threads, but with a great number of HW threads , show their lacks. Examples of this category are

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

MERGING ALGORITHMS
Divide the data in parts, and each part is sorted by a thread. When the parts are sorted, must merge them for obtain the final results. The problem of these algorithms is they need additional memory for the merge, and usually with the same size than the data.

With a small number of threads, have similar speed than the subdivision algorithms, but with many threads they are much more 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 a non stable parallel sort algorithm, create for processors connected with shared memory. Provide an excellent performance in machines with many HW threads, similar to the GCC Parallel Sort , and better than TBB, with the additional advantage of the small memory consumption.

This algorithm use as auxiliary memory a 1024 elements buffer for each thread. The worst case for the algorithm is when have very big elements and 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

** This algorithm had been ideate, designed and implemented beginning from zero. After read hundreds of articles and books, I didn't find any similar. If someone knows something about this or something similar, please, say me.*

Anyway, the important is not the author, is provide a simple, fast and robust algorithm to the community of programmers.

INDEX

1.- OVERVIEW OF THE SORTING PARALLEL ALGORITHMS

2.- INTERNAL DESCRIPTION OF THE ALGORITHM
2.1.- SUBDIVISION OF THE MERGE
2.2.- NUMBER OF ELEMENTS NO MULTIPLE OF THE BLOCK SIZE.
2.3.- INDIRECT SORTING
2.4.- BLOCK SORT FROM A INDEX
2.5- SEQUENCE PARALLELIZATION

3.- BENCHMARKS
3.1.- INTRODUCTION
3.2.- DESCRIPTION
3.3.- NUMBERS
3.4.- STRINGS
3.5.- OBJECTS

4.- BIBLIOGRAPHY

5.- GRATITUDE

1.- OVERVIEW OF THE SORTING PARALLEL ALGORITHMS

Now, in the non stable parallel sorting algorithms, we can find basically two kinds of algorithms:

1.- SUBDIVISION ALGORITHMS

As Parallel Quick Sort. One thread divide the problem in two parts. Each part obtained is divided by other thread, until the subdivision generate sufficient parts for to have busy all the threads. By example a 32 HW treads 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

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 hight.

2.- MERGING ALGORITHMS,

Divide the data in parts, for to be sorted separately by the threads. And after, the sorted parts must be merged. But the main problem, of these algorithms is the memory used in the merge, usually of the same size than the data.

These algorithms provide the best performance with many threads, but their performance with a low number of threads is poor, being surpassed by the subdivision algorithms.

2.- INTERNAL DESCRIPTION OF THE ALGORITHM

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

This new algorithm only need an auxiliary memory of 1024 elements for each HW thread. The worst case 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

The secret is to use algorithms, which with only 1 thread are not the fastest, but are easily parallelized, and need only a small auxiliary memory for to do the operations.

The algorithm divide the number of elements in a number of parts as the first power of two, equal or great than the number of threads to use. (By example: with 3 threads, make 4 parts, for 12 threads make 16 parts…) Each part obtained is sorted by the parallel intro sort algorithm.

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

With the sorted parts, begin the merge of pair of parts.

We consider the elements are in blocks of fixed size. For to begin, we establish a restriction, which will be removed later. The number of elements to sort (N) is 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 for to sort, and when finish 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 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
44 46 50 71	36 38 45 60	16 20 27 29				28 29 32 34	28 29 32 34
		35 37 39 40	35 36 37 38				35 36 37 38
		36 38 45 60	39 40 45 60	39 40 44 45			39 40 44 45
		44 46 50 71		46 50 60 71			46 50 60 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, for to be executed by different threads.

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

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

In the option 1, the frontier is between the blocks 4 and 5, and the last of th 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.

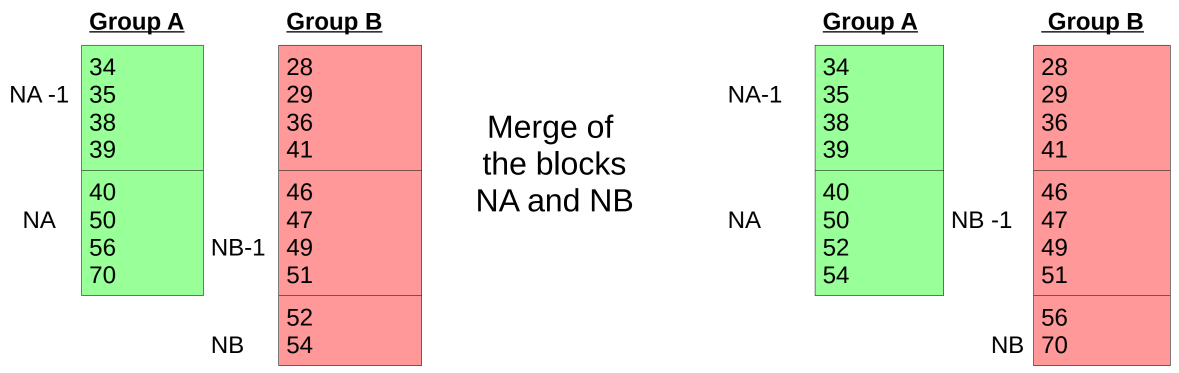
2.2.- NUMBER OF ELEMENTS NOT MULTIPLE OF THE BLOCK SIZE

Until now, we consider the number of elements 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. For to manage this block, we do a easy operation, described in the next example :

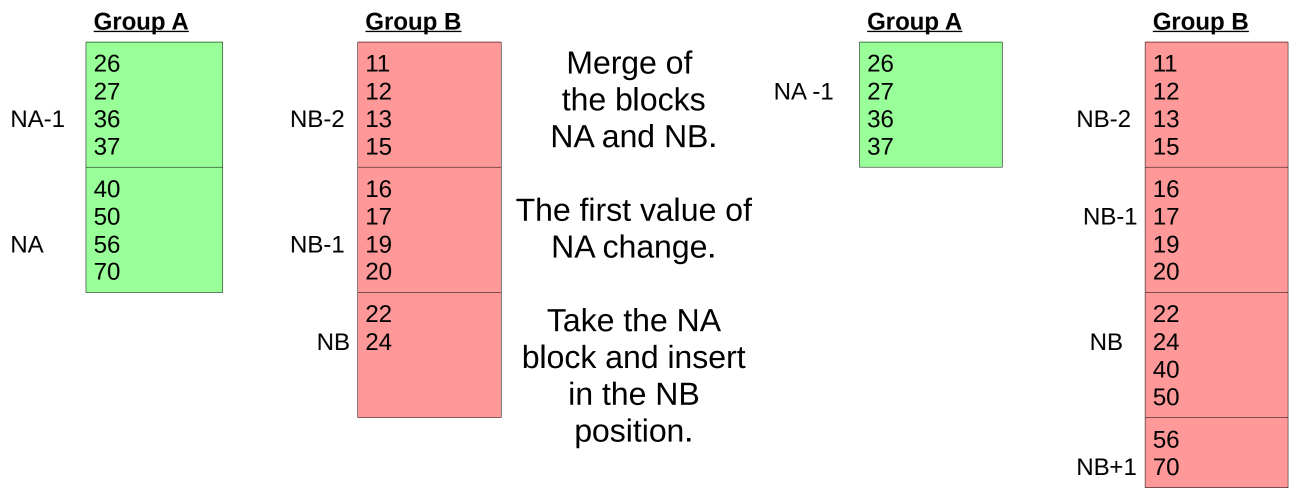
We have two groups of blocks A and B, with NA and NB blocks respectively. The tail block, if exist, always is in the group B

Merge the tail block with the last block of the group A. With the merge appear two cases showed 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 the today's computers, the bottleneck is the data bus. When the process need to manage many memory, as in the parallel sorting algorithms, the data bus limit the speed of the algorithm.

In the benchmarks done on a 32 HE threads, sorting N elements of 64 Bytes of size, need the same time than sort 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, for to be merged in the next step. This slow down the algorithm, due to the previously commented

For to avoid this, do an indirect merge. We have an index with the relative position of the blocks. This imply the blocks to merge are not contiguous. This idea don't change the validity of the algorithm.

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

2.4.- BLOCK SORTING FROM A INDEX

The tail block, if exist, is always in the last position, and don'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 have too, an index (I), which is a vector with the relative position of the elements of D

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

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

For to do the movement, we need an auxiliary variable (Aux) of the same size than the data. In this example is a number, in the case of blocks, the variable must have the size of a block.

This can be a bit confusing. For to easy understand, describe with words 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 in the position 0 of the index, and 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 is the position 2.
- When doing this successively, and when 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.

When use a position of the index, we write this position, with their position. This is, after use the position 0 of the index, write a 0 in the position 0, and after use the position 4, write 4 in this position.

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 close cycle. This cycle have a sequence formed by the position of the elements passed. In this example is 0, 4, 2, 5, 6, 7, 3, 1. From the sequence, the movement of the data is trivial

With small elements the sequence is usefulness, because, instead extract the sequence, mode 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.

In an index, can appear several cycles, with their corresponding sequences. For 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 actualize 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 permit to be done in parallel, but it's not very good, because a long sequence maintain busy a thread, and the others threads are waiting , because they are finished with the sort sequences. Or even worst, there is only one sequence.

The good new is 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, and for to explain, we use 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

In this example, we can see all the described:

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

For the benchmark, 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 have all the code, instructions and procedures in https://github.com/fjtapia/sort_parallel_benchmark

For the comparison, we use the next parallel algorithms:

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

3.2.- DESCRIPTION

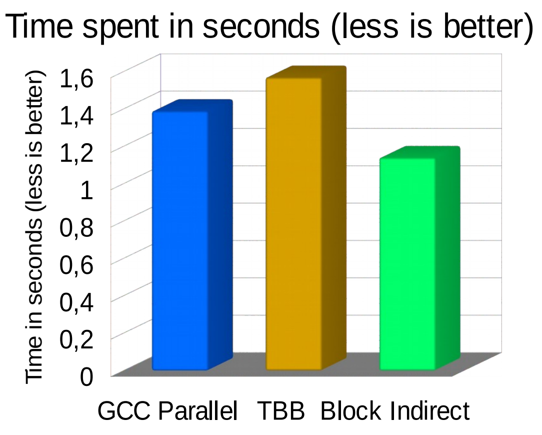
This benchmark had been done with a Dell Power Edge R520 12 G 2.1 GHz with two Intel Xeon E5-2690, with the Hyper Threading activate and with 32 HW threads.
The compiler used was the GCC 5.2 64 bits

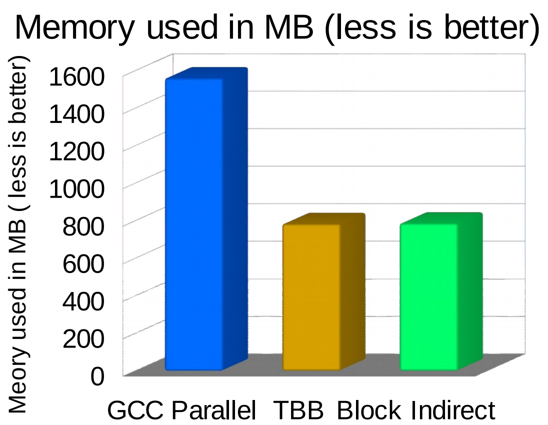
The benchmark have 3 parts:

- 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

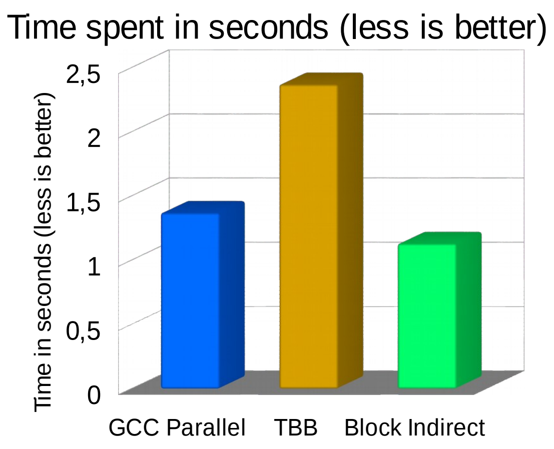
Sorting of 100 000 000 random 64 bits numbers.

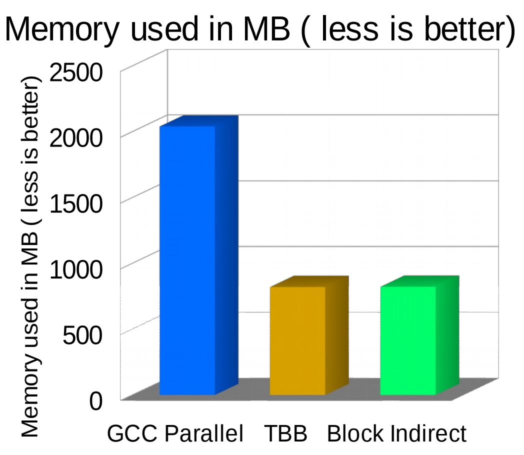




3.4.- STRINGS

Sorting 10 000 000 strings, randomly filled



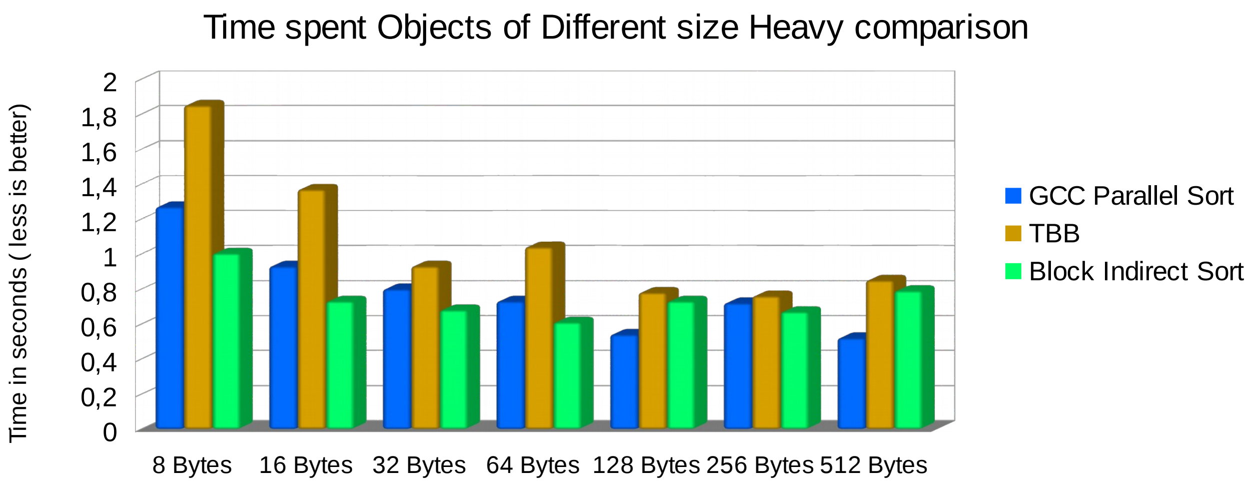


3.5.- OBJECTS

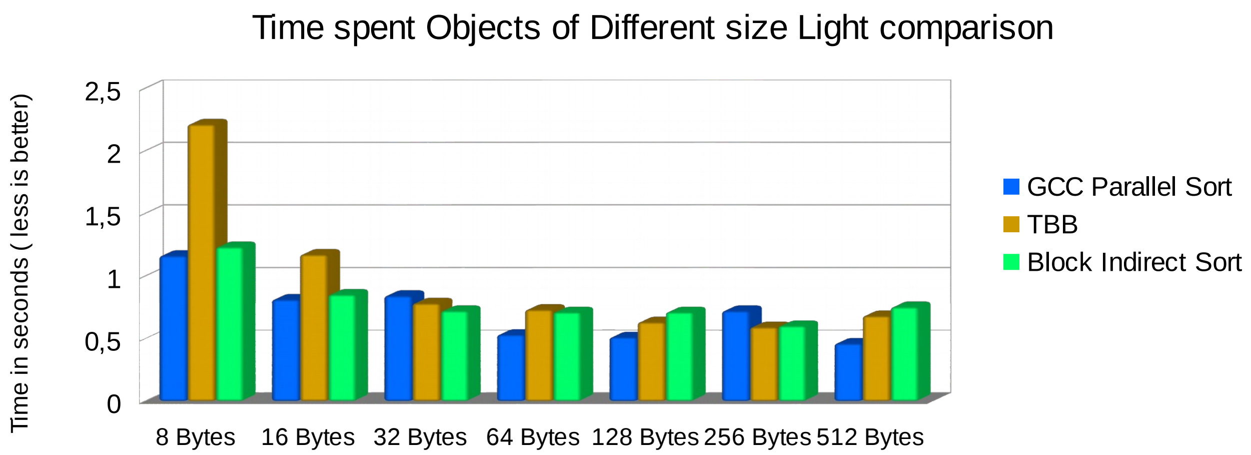
The objects are arrays of 64 bits numbers, randomly filled. An 8 byte object is an array of 1 element, a 16 bytes object have 2 numbers and successively , until the object of 512 bytes, which have 64 numbers.

The benchmark had been done using two kind of comparison:

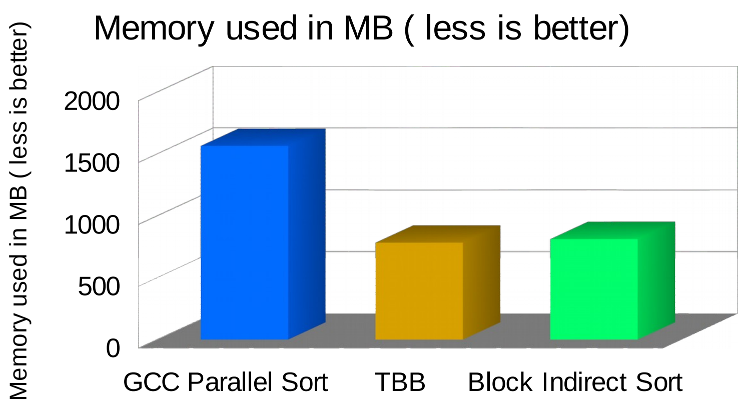
1.- Heavy comparison. The comparison is the sum of all the numbers in the array. In each comparison, the sum is done



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.