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 frecuently used in the filter and sorting is quicksort.

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

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.- BENCHMARKS

2.1.- INTRODUCTION

For the bechmark, 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, intructions 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

2.2.- DESCRIPTION

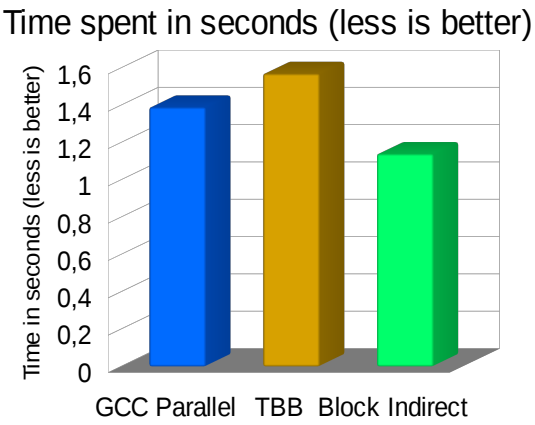
This benchmark had been done with a Dell Power Edge R520 12 G 2.1 GHz with two Intel Xeon E5-2690, with the Hyper Threading activate and with 32 HW threads.
The compiler used was the GCC 5.2 64 bits

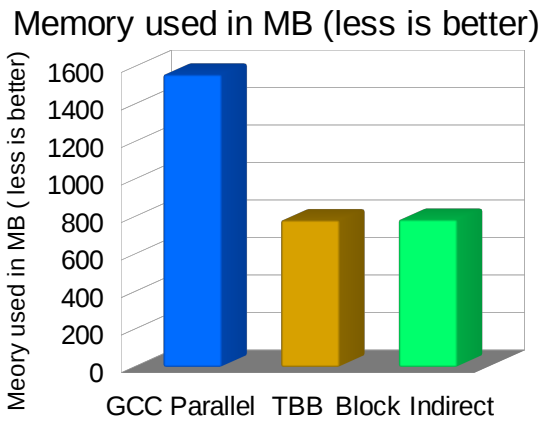
The benchmark have 3 parts:

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

2.3.- NUMBERS

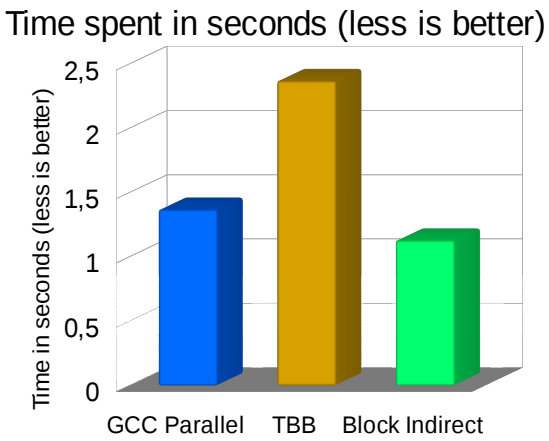
Sorting of 100 000 000 random 64 bits numbers.

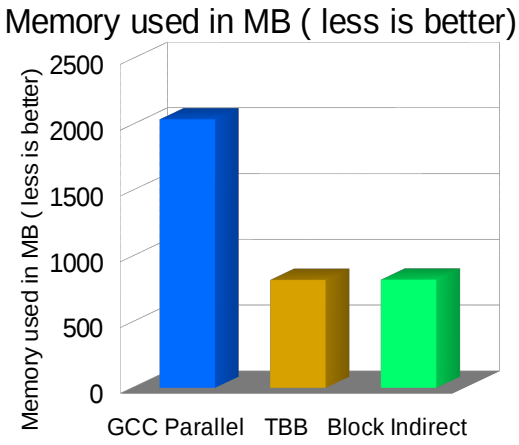




2.4.- STRINGS

Sorting 10 000 000 strings, randomly filled



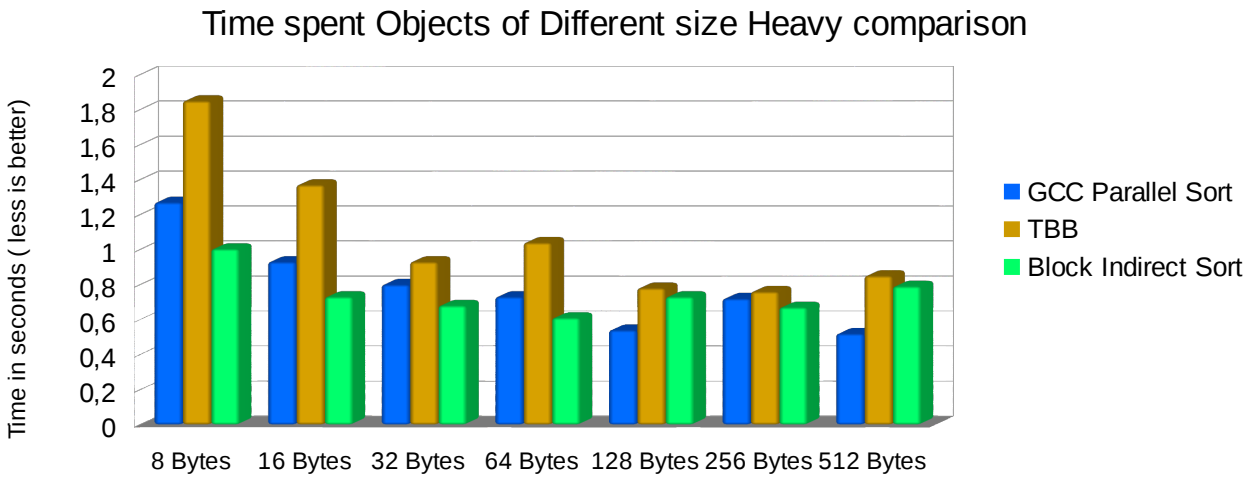


2.5.- OBJECTS

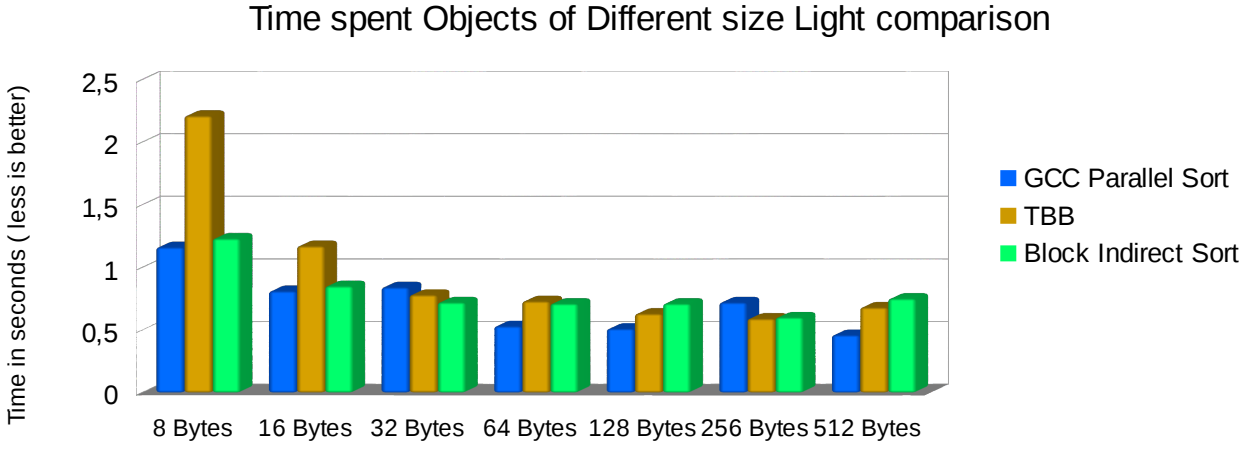
The objects are arrays of 64 bits numbers, randomly filled. An 8 byte object is an array of 1 element, a 16 bytes object have 2 numbers and successively , until the object of 512 bytes, which have 64 numbers.

The benchmark had been done using two kind of comparison:

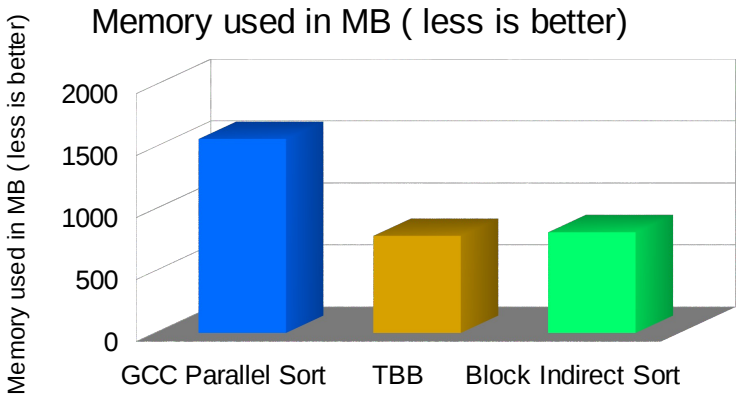
1.- Heavy comparison. The comparison is the sum of all the numbers in the array. In each comparison, the sum is done



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



3.- Memory used



3.- 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)

4.- 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.