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.

This make appear algorithms, that until now was exclusive of the big computers. Now they must simplify their use, because the users, are not the highly qualified programmers of the big systems, are a big group of programmers.

This take to the revision of many algorithms, which are good with a small number of threads, but, show their lacks with a great number of HW threads.

This generate an undesirable duality in many parallel algorithms. With a small number of threads use one algorithm, and with a big number use other. Then the SW designed for a small machine is inadequate for a big machine and vice versa. This is the case of the parallel sorting algorithms.

This new algorithm is a non stable parallel sort algorithm, create for processors connected with shared memory. Provide an excellent performance in small and big machines, deleting the duality, 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 fast, robust, and easy to use 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 based on Quick Sort,
As Threading Building Blocks (TBB). 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 which the first division are the hardest, because the number of elements is very hight.

2.- Algorithms cutting the problem,
As the Parallel Sort of GCC. Generate work for all the HW threads, simultaneously, but to cut the problem is a hard work, and it performance with a low number of threads is poor, being surpassed by the subdivision algorithms.

But the main problem, of these algorithms is the memory used. For to do the subdivision, use an additional memory, usually of the same size than the data

This new algorithm (Block Indirect), cut the problem, with similar performance than the GCC Parallel Sort with many HW threads, but using a memory as 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

1.1.- FUTURE CONSIDERATIONS

In future versions, the algorithm will have several improvements in the parallelization of the parts, and in the implementation, with the use of SW tools which simplify the design and improve the speed.

This version had been designed for a shared memory system. But the ideas and concepts, simplify the design for a system connected with a network, optimizing the use of the memory in the nodes and minimizing the information moved between the nodes of the network, for to achieve the optimal performance.

2.- BENCHMARKS

2.1.- INTRODUCTION

For the comparison, we use the next parallel algorithms:

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

The Block Indirect is hybrid algorithm. When the number of HW threads is greater than 8 use the new algorithm, and in the other cases use the parallel introsort. The algorithm decide which algorithm to use dynamically. The same program in a small machine use an algorithm , and in a big machine use other.

In machines with small number of cores, TBB is much faster than GCC parallel. With 8 HW threads the speeds are similar. But in machines with many cores, GCC parallel can be 40% faster than TBB.

TBB have an excellent implementation, but with many cores is slower than the GCC parallel, due to their internal algorithm.

Block Indirect, due to their hybrid nature, when run in a small machine, have similar performance than TBB, and in a big machine have similar performance than GCC parallel, but without their big auxiliary memory.

The cache size, improve all the algorithms identically, and although is very important, don't affect to the relative position in the performance of the algorithms.

In the processors with HT activate, usually Block Indirect is faster than GCC parallel, and with HT deactivate, GCC parallel is faster than Block Indirect

The improvement in the data bus, usually make faster GCC and TBB than Block Indirect.
When the size of the data is big, generate a saturation in the data bus, and in many processors the times of GCC, TBB and Block Indirect are similar

2.2.- DESCRIPTION

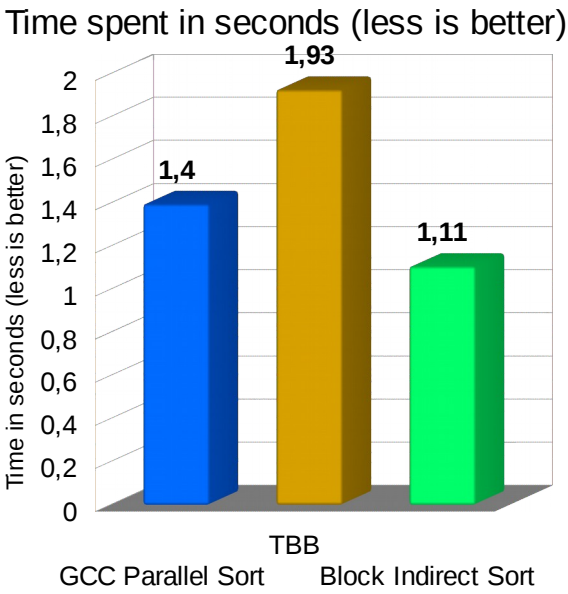
This benchmark had been done with a Dell Power Edge R520 12 G 2.1 GHz with two Intel Xeon E5-2690, with the Hyper Threading activate and with 32 HW threads.
The compiler used was the GCC 4.9 64 bits

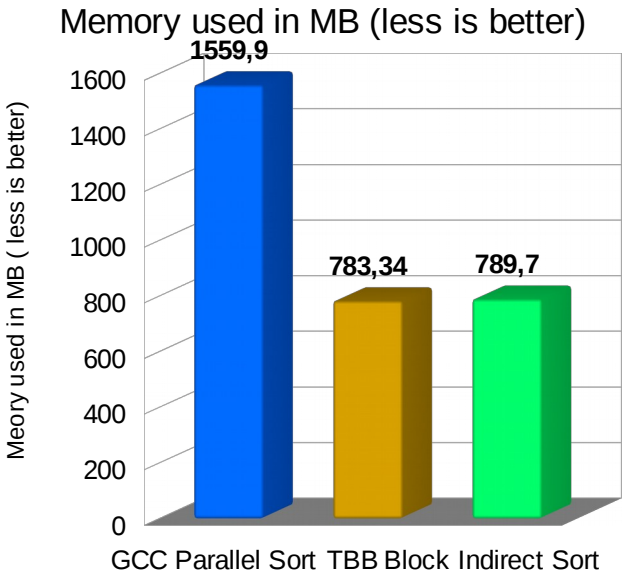
The benchmark have 3 parts:

- 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

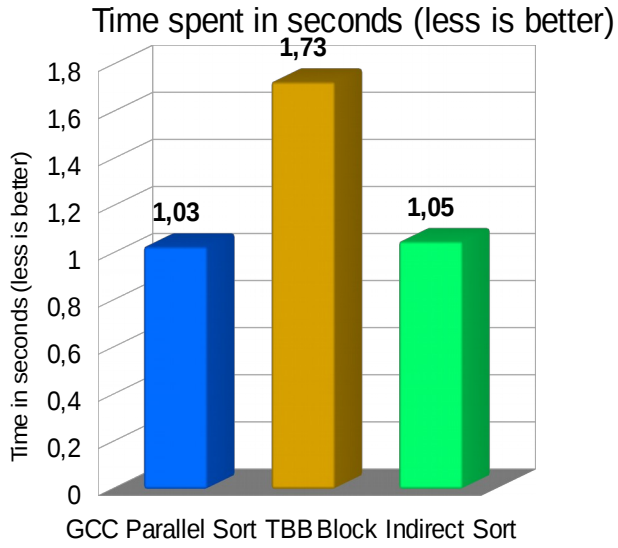
Sorting of 100 000 000 random 64 bits numbers.

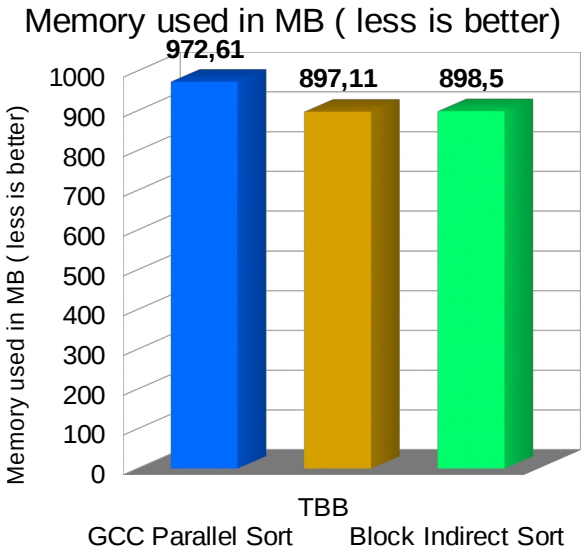




2.4.- STRINGS

Sorting 10 000 000 strings, randomly filled



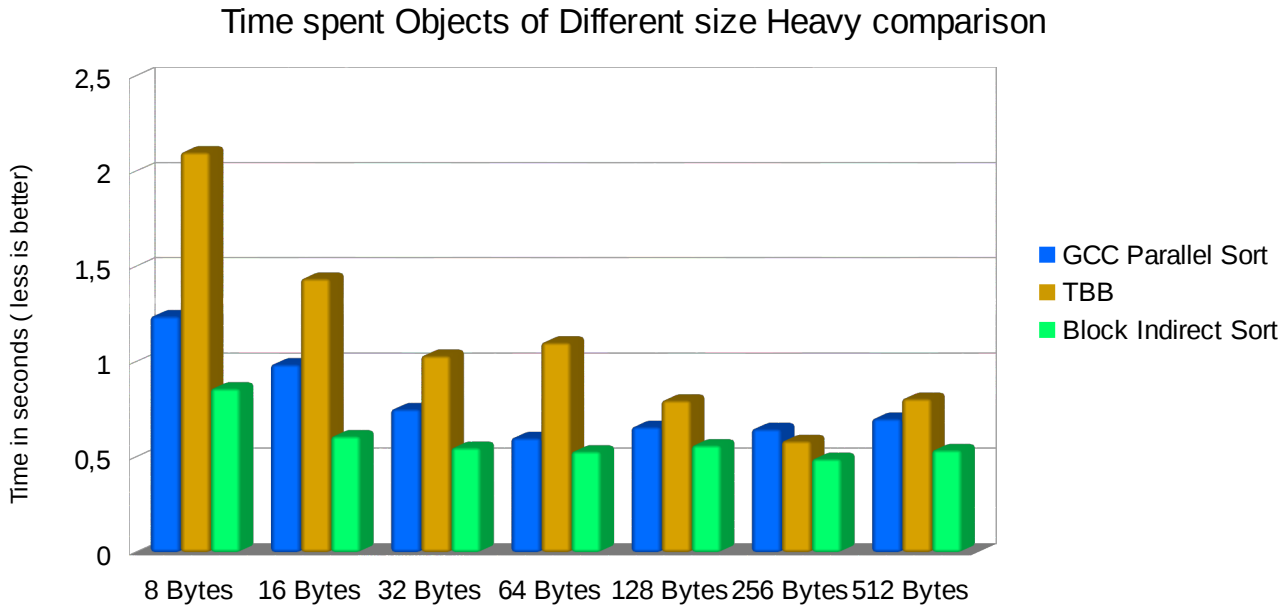


2.5.- OBJECTS

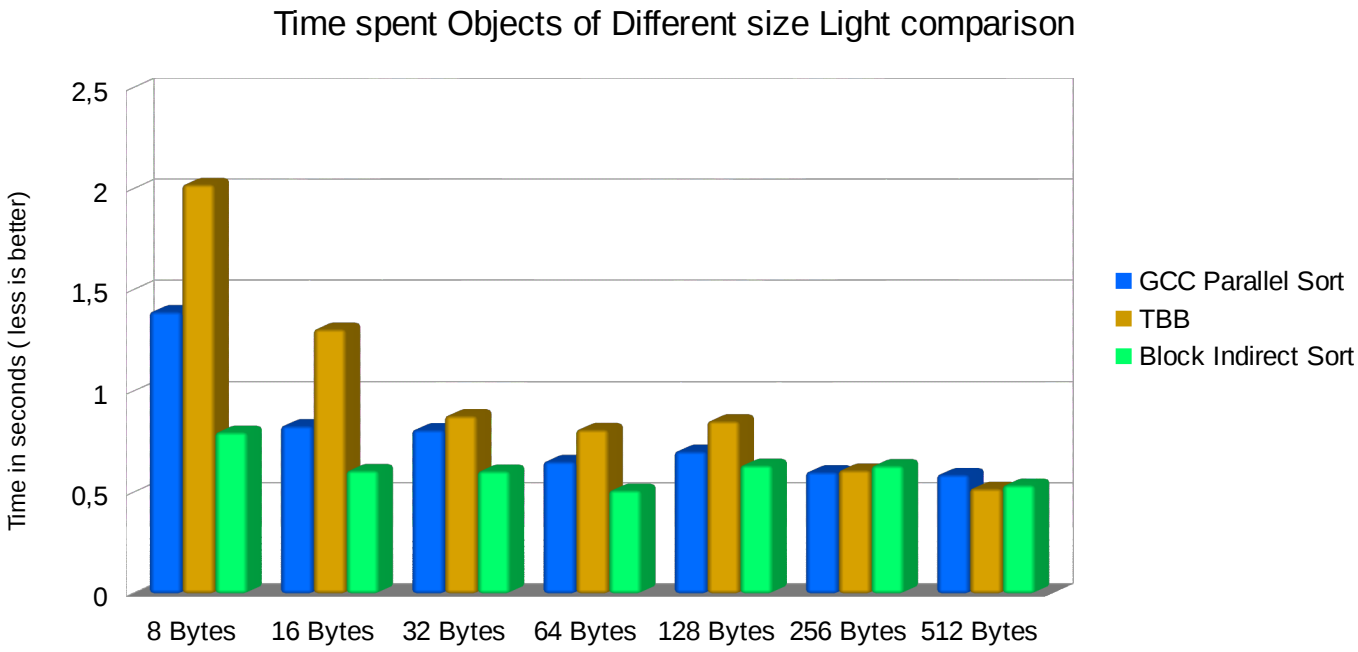
The objects are arrays of 64 bits numbers, randomly filled. An 8 byte object is an array of 1 element, a 16 bytes object have 2 numbers and successively , until the object of 512 bytes, which have 64 numbers.

The benchmark had been done using two kind of comparison:

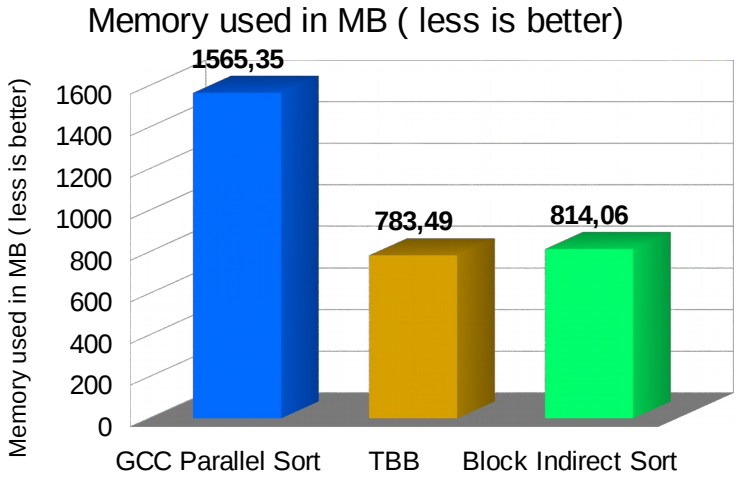
1.- Heavy comparison. The comparison is the sum of all the numbers in the array. In each comparison, the sum is done



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.

ANNEX A

This is the file generated by the benchmark program:

```
************************************************************
**                                                        **
**                  B E N C H M A R K                     **
**                                                        **
************************************************************
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
```

```
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 45
Stepping:              7
CPU MHz:               2099.928
BogoMIPS:              4200.80
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31

  100000000 uint64_t elements randomly filled
================================================
GCC parallel sort            : 1.39747 secs
TBB parallel_sort            : 1.92832 secs
Block Indirect sort          : 1.10781 secs

  10000000 strings randomly filled
================================================
GCC parallel sort            : 1.03054 secs
TBB parallel_sort            : 1.73018 secs
Block Indirect sort          : 1.05267 secs

================================================================
=                OBJECT COMPARISON                             =
=              ----------------------                          =
= The objects are arrays of 64 bits numbers                    =
= They are compared in two ways :                              =
=    (H) Heavy : The comparison is the sum of all the numbers  =
=                of the array                                  =
=    (L) Light : The comparison is with the first element of   =
=                the array, as a key                           =
================================================================

100000000 elements of size 8 randomly filled
============================================

  H E A V Y   C O M P A R I S O N
==================================
GCC parallel sort            : 1.24054 secs
TBB parallel_sort            : 2.10116 secs
Block Indirect sort          : 0.861248 secs

  L I G H T   C O M P A R I S O N
====================================
GCC parallel sort            : 1.39412 secs
TBB parallel_sort            : 2.0232 secs
Block Indirect sort          : 0.797875 secs


50000000 elements of size 16 randomly filled
============================================

  H E A V Y   C O M P A R I S O N
==================================
GCC parallel sort            : 0.987663 secs
TBB parallel_sort            : 1.43775 secs
Block Indirect sort          : 0.610134 secs

  L I G H T   C O M P A R I S O N
====================================
GCC parallel sort            : 0.829797 secs
TBB parallel_sort            : 1.30786 secs
Block Indirect sort          : 0.608633 secs


25000000 elements of size 32 randomly filled
============================================
```

```
  H E A V Y   C O M P A R I S O N
==================================
GCC parallel sort            : 0.752434 secs
TBB parallel_sort            : 1.03325 secs
Block Indirect sort          : 0.550265 secs

  L I G H T   C O M P A R I S O N
====================================
GCC parallel sort            : 0.810079 secs
TBB parallel_sort            : 0.879593 secs
Block Indirect sort          : 0.605998 secs


12500000 elements of size 64 randomly filled
============================================

  H E A V Y   C O M P A R I S O N
==================================
GCC parallel sort            : 0.601689 secs
TBB parallel_sort            : 1.1025 secs
Block Indirect sort          : 0.530544 secs

  L I G H T   C O M P A R I S O N
====================================
GCC parallel sort            : 0.653337 secs
TBB parallel_sort            : 0.811457 secs
Block Indirect sort          : 0.510883 secs


6250000 elements of size 128 randomly filled
============================================

  H E A V Y   C O M P A R I S O N
==================================
GCC parallel sort            : 0.659467 secs
TBB parallel_sort            : 0.798281 secs
Block Indirect sort          : 0.562632 secs

  L I G H T   C O M P A R I S O N
====================================
GCC parallel sort            : 0.703836 secs
TBB parallel_sort            : 0.853894 secs
Block Indirect sort          : 0.634528 secs


3125000 elements of size 256 randomly filled
============================================

  H E A V Y   C O M P A R I S O N
==================================
GCC parallel sort            : 0.648604 secs
TBB parallel_sort            : 0.585702 secs
Block Indirect sort          : 0.491528 secs

  L I G H T   C O M P A R I S O N
====================================
GCC parallel sort            : 0.601979 secs
TBB parallel_sort            : 0.610897 secs
Block Indirect sort          : 0.632575 secs


1562500 elements of size 512 randomly filled
============================================

  H E A V Y   C O M P A R I S O N
==================================
GCC parallel sort            : 0.702802 secs
TBB parallel_sort            : 0.80763 secs
Block Indirect sort          : 0.538038 secs

  L I G H T   C O M P A R I S O N
====================================
GCC parallel sort            : 0.589143 secs
TBB parallel_sort            : 0.519865 secs
Block Indirect sort          : 0.536712 secs
```