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 block_size elements buffer for each thread. The block_size is an internal parameter of the algorithm, which, in order to achieve the highest speed, change according the size of the objects to sort according the next table. The strings use a block_size of 128.

object size	1 - 15	16 - 31	32 - 63	64 - 127	128 - 255	256 - 511	512 -
block_size	4096	2048	1024	768	512	256	128

The worst case memory usage for the algorithm is when elements are large and there are many threads. With big elements (512 bytes), and 12 threads, the memory measured was:

- GCC Parallel Sort 1565 MB
- Threading Building Blocks (TBB) 783 MB
- Block Indirect Sort 812 MB

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.- *BENCHMARKS*

2.1.- 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

2.2.- DESCRIPTION

The benchmark are running in a machine with a I7 5820 3.3 GHz 6 cores, 12 threads, quad channel memory (2133 MHz) with Ubuntu and the GCC 5.2 compiler

The compiler used was the GCC 5.2 64 bits

The benchmark have 3 parts:

1.- Sort of 100000000 uint64_t numbers randomly generated. The utility of this benchmark is to see the speed with small elements with a very fast comparison.

2.- Sort of 10000000 of strings randomly filled. The comparison is no so easy as the integers.

3.- Sort of objects of several sizes. The objects are arrays of 64 bits numbers, randomly filled. We will check with arrays of 1 , 2 , 4, 8, 16, 32 and 64 numbers.

Definition of the object	Bytes	Number of elements to sort
uint64_t [1]	8	100 000 000
uint64_t [2]	16	50 000 000
uint64_t [4]	32	25 000 000
uint64_t [8]	64	12 500 000
uint64_t [16]	128	6 250 000
uint64_t [32]	256	3 125 000
uint64_t [64]	512	1 562 500

The C++ definition of the objects is

```
template <uint32_t NN>
struct int_array
{   uint64_t M[NN];
};
```

The comparison between objects can be of two ways:

- Heavy comparison : The comparison is done with the sum of all the numbers of the array. In each comparison, make the sum.
- Light comparison : It's done using only the first number of the array, as a key in a register.

2.3.- LINUX 64 GCC 5.2 Benchmarks

The benchmark are running in a I7 5820 3.3 GHz 6 cores, 12 threads, quad channel memory (2133 MHz) with Ubuntu and the GCC 5.2 compiler.

2.3.1.-SINGLE THREAD ALGORITHMS

The algorithms involved in this benchmark are :

	Stable	Memory used	Comments
GCC sort	no	N + Log N	
boost sort	no	N + Log N	
GCC stable_sort	yes	N + N / 2	
Boost stable_sort	yes	N + N / 2	
Boost spreadsort	yes	N + Log N	Extremely fast algorithm, only for integers, floats and strings

INTEGER BENCHMARKS Sort of 100000000 64 bits numbers, randomly filled

	Time	Memory
GCC sort	8.33 secs	784 MB
Boost sort	8.11 secs	784 MB
GCC stable sort	8.69 secs	1176 MB
Boost stable sort	8.75 secs	1175 MB
Boost Spreadsort	4.33 secs	784 MB

STRINGS BENCHMARKS Sort of 10 000 000 strings randomly filled

	Time	Memory
GCC sort	6.39 secs	820 MB
Boost sort	7.01 secs	820 MB
GCC stable sort	12.99 secs	1132 MB
Boost stable sort	9.17 secs	976 MB
Boost Spreadsort	2.44 secs	820 MB

OBJECTS BENCHMARKS Sorting of objects of different sizes. The objects are arrays of 64 bits numbers. This benchmark is done using two kinds of comparison.

Heavy comparison : The comparison is done with the sum of all the numbers of the array. In each comparison, make the sum.

	8 bytes	16 bytes	32 bytes	64 bytes	128 bytes	256 bytes	512 bytes	Memory used
GCC sort	8.75	4.49	3.03	1.97	1.71	1.37	1.17	783 MB
Boost sort	8.19	4.42	2.65	1.91	1.67	1.35	1.09	783 MB
GCC stable_sort	10.23	5.67	3.67	2.94	2.6	2.49	2.34	1174 MB
Boost stable_sort	8.85	5.11	3.18	2.41	2.01	1.86	1.60	1174 MB

Light comparison : It's done using only the first number of the array, as a key in a register.

	8 bytes	16 bytes	32 bytes	64 bytes	128 bytes	256 bytes	512 bytes	Memory used
GCC sort	8.69	4.31	2.35	1.50	1.23	0.86	0.79	783 MB
Boost sort	8.18	4.04	2.25	1.45	1.24	0.88	0.76	783 MB
GCC stable_sort	10.34	5.26	3.20	2.57	2.47	2.41	2.30	1174 MB
Boost stable_sort	8.92	4.59	2.51	1.94	1.68	1.68	1.50	1174 MB

2.3.2.-PARALLEL ALGORITHMS

The algorithms involved in this benchmark are :

	Stable	Memory used	Comments
GCC parallel sort	No	2N	Based on OpenMP
TBB parallel sort	No	N + LogN	
Boost parallel sort	No	N +block_size*num threads	New parallel algorithm
GCC parallel stable sort	Yes	2 N	Based on OpenMP
Boost parallel stable sort	Yes	N / 2	
Boost sample sort	Yes	N	
TBB parallel stable sort	Yes	N	Experimental code, not in the TBB official

The block_size is an internal parameter of the algorithm, which in order to achieve the highest speed, change according the size of the objects to sort according to the next table. The strings use a block_size of 128.

object size (bytes)	1 - 15	16 - 31	32 - 63	64 - 127	128 - 255	256 - 511	512 -
block_size	4096	2048	1024	768	512	256	128

For the benchmark I use the next additional code:

- Threading Building Blocks (TBB)
- OpenMP
- Threading Building Block experimental code (https://software.intel.com/sites/default/files/managed/48/9b/parallel_stable_sort.zip)

The most significant of this parallel benchmark is the comparison between the Parallel Sort algorithms. GCC parallel sort is extremely fast with many cores, but need an auxiliary memory of the same size then the data. In the other side Threading Building Blocks (TBB), is not so fast with many cores , but the auxiliary memory is LogN.

The Boost Parallel Sort (internally named Block Indirect Sort), is a new algorithm created and implemented by the author for this library, which combine the speed of GCC Parallel sort, with a small memory consumption (block_size elements for each thread). The worst case for this algorithm is when have very big elements and many threads. With big elements (512 bytes), and 12 threads, The memory measured was:

GCC Parallel Sort (OpenMP)	1565 MB
Threading Building Blocks (TBB)	783 MB
Block Indirect Sort	812 MB

In machines with a small number of HW threads, TBB is faster than GCC, but with a great number of HW threads GCC is more faster than TBB. Boost Parallel Sort have similar speed than GCC Parallel Sort with a great number of HW threads, and similar speed to TBB with a small number.

INTEGER BENCHMARKS Sort of 100 000 000 64 bits numbers, randomly filled

	time (secs)	**memory (MB)**
OMP parallel_sort	1,25	1560
TBB parallel_sort	1,64	783
Boost parallel_sort	1,08	786
OMP parallel_stable_sort	1,56	1948
TBB parallel_stable_sort	1,56	1561
Boost sample_sort	1,19	1565
Boost parallel_stable_sort	1,54	1174

STRING BENCHMARK Sort of 10000000 strings randomly filled

	time (secs)	**memory (MB)**
OMP parallel_sort	1,49	2040
TBB parallel_sort	1,84	820
Boost parallel_sort	1,3	822
OMP parallel_stable_sort	2,25	2040
TBB parallel_stable_sort	2,1	1131
Boost sample_sort	1,51	1134
Boost parallel_stable_sort	2,1	977

OBJECT BENCHMARKS Sorting of objects of different sizes. The objects are arrays of 64 bits number. This benchmark is done using two kinds of comparison.

Heavy comparison : The comparison is done with the sum of all the numbers of the array. In each comparison, make the sum.

	8 bytes	**16 bytes**	**32 bytes**	**64 bytes**	**128 bytes**	**256 bytes**	**512 bytes**	**Memory Used**
OMP parallel_sort	1,27	0,72	0,56	0,45	0,41	0,39	0,32	1565
TBB parallel_sort	1,63	0,8	0,56	0,5	0,44	0,39	0,32	783
Boost parallel_sort	1,13	0,67	0,53	0,47	0,43	0,41	0,34	812
OMP parallel_stable_sort	1,62	1,38	1,23	1,19	1,09	1,07	0,97	1954
TBB parallel_stable_sort	1,58	1,02	0,81	0,76	0,73	0,73	0,71	1566
Boost sample_sort	1,15	0,79	0,63	0,62	0,62	0,61	0,6	1566
Boost parallel_stable_sort	1,58	1,02	0,8	0,76	0,73	0,73	0,71	1175

Light comparison : It's done using only the first number of the array, as a key in a register.

	8 bytes	16 bytes	32 bytes	64 bytes	128 bytes	256 bytes	512 bytes	Memory used
OMP parallel_sort	1,24	0,71	0,48	0,41	0,38	0,35	0,32	1565
TBB parallel_sort	1,66	0,8	0,52	0,43	0,4	0,35	0,32	783
Boost parallel_sort	1,11	0,65	0,49	0,43	0,41	0,37	0,34	812
OMP parallel_stable_sort	1,55	1,36	1,23	1,18	1,09	1,07	0,97	1954
TBB parallel_stable_sort	1,58	0,91	0,75	0,72	0,71	0,72	0,71	1566
Boost parallel_stable_sort	1,16	0,74	0,63	0,62	0,61	0,61	0,6	1566
Boost sample_sort	1,56	0,91	0,75	0,72	0,72	0,72	0,71	1175

2.4.- WINDOWS 10 VISUAL STUDIO 2015 x64 Benchmarks

The benchmark are running in a virtual machine with Windows 10 and 10 threads over a I7 5820 3.3 GHz with Visual Studio 2015 C++ compiler.

2.4.1.-SINGLE THREAD ALGORITHMS

The algorithms involved in this benchmark are :

	Stable	Memory used	Comments
std::sort	no	N + Log N	
boost sort	no	N + Log N	
std::stable_sort	yes	N + N / 2	
Boost stable_sort	yes	N + N / 2	
Boost spreadsort	yes	N + Log N	Extremely fast algorithm, only for integers, floats and strings

INTEGER BENCHMARKS Sort of 100000000 64 bits numbers, randomly filled

	Time (secs)	Memory (MB)
std::sort	13	763
Boost sort	10,74	763
std::stable_sort	14,94	1144
Boost stable_sort	13,37	1144
Boost spreadsort	9,58	763

STRING BENCHMARKS Sort of 10 000 000 strings randomly filled

	Time (secs)	Memory (MB)
std::sort	13,3	862
Boost sort	13,6	862
std::stable_sort	26,99	1015
Boost stable_sort	20,64	1015
Boost spreadsort	5,7	862

OBJECTS BENCHMARK Sorting of objects of different sizes. The objects are arrays of 64 bits numbers. This benchmark is done using two kinds of comparison.

Heavy comparison : The comparison is done with the sum of all the numbers of the array. In each comparison, make the sum.

	8 bytes	***16 bytes***	***32 bytes***	***64 bytes***	***128 bytes***	***256 bytes***	***512 bytes***	***Memory used***
std::sort	13,36	6,98	4,2	2,58	2,87	2,37	2,29	763
Boost sort	10,54	5,61	3,26	2,72	2,45	1,76	1,73	763
std::stable_sort	15,49	8,47	5,47	3,97	3,85	3,55	2,99	1144
Boost stable_sort	13,11	8,86	5,06	4,16	3,9	3,06	3,32	1144

Light comparison : It's done using only the first number of the array, as a key in a register.

	8 bytes	***16 bytes***	***32 bytes***	***64 bytes***	***128 bytes***	***256 bytes***	***512 bytes***	***Memory used***
std::sort	14,15	7,26	4,33	2,69	1,92	1,98	1,73	763
Boost sort	10,33	5	2,99	1,85	1,53	1,46	1,4	763
std::stable_sort	14,68	7,64	4,29	3,33	3,22	2,86	3,08	1144
Boost stable_sort	13,59	8,36	4,45	3,73	3,16	2,81	2,6	1144

2.4.2.-PARALLEL ALGORITHMS

The algorithms involved in this benchmark are :

	Stable	**Memory used**	**Comments**
PPL parallel sort	No	N	
PPL parallel buffered sort	No	2 N	
Boost parallel sort	No	N +block_size*num threads	New parallel algorithm
Boost parallel stable sort	Yes	N + N / 2	
Boost sample sort	Yes	2 N	

The block_size is an internal parameter of the algorithm, which in order to achieve the highest speed, change according the size of the objects to sort according to the next table. The strings use a block_size of 128.

object size (bytes)	1 - 15	16 - 31	32 - 63	64 - 127	128 - 255	256 - 511	512 -
block_size	4096	2048	1024	768	512	256	128

INTEGER BENCHMARKS Sort of 100 000 000 64 bits numbers, randomly filled

	Time (secs)	**Memory (MB)**
PPL parallel sort	3,11	764
PPL parallel buffered sort	1,74	1527
Boost parallel sort	2,1	764
Boost sample sort	2,78	1511
Boost parallel stable sort	3,3	1145

STRINGS BENCHMARK Sort of 10000000 strings randomly filled

	Time (secs)	Memory (MB)
PPL parallel sort	3,76	864
PPL parallel buffered sort	3,77	1169
Boost parallel sort	3,41	866
Boost sample sort	3,74	1168
Boost parallel stable sort	5,7	1015

OBJECTS BENCHMARKS Sorting of objects of different sizes. The objects are arrays of 64 bits number. This benchmark is done using two kinds of comparison.

Heavy comparison : The comparison is done with the sum of all the numbers of the array. In each comparison, make the sum.

	8 bytes	*16 bytes*	*32 bytes*	*64 bytes*	*128 bytes*	*256 bytes*	*512 bytes*	*Memory used*
PPL parallel sort	2,84	1,71	1,01	0,84	0,89	0,77	0,65	764
PPL parallel buffered sort	2,2	1,29	2	0,88	0,98	1,32	0,82	1527
Boost parallel sort	1,93	0,82	0,9	0,72	0,77	0,68	0,69	764
Boost sample sort	3,02	2,03	2,15	1,41	1,55	1,82	1,39	1526
Boost parallel stable sort	3,36	2,67	1,62	1,45	1,38	1,19	1,37	1145

Light comparison : It's done using only the first number of the array, as a key in a register.

	8 bytes	*16 bytes*	*32 bytes*	*64 bytes*	*128 bytes*	*256 bytes*	*512 bytes*	*Memory used*
PPL parallel sort	3,1	1,37	0,97	0,7	0,61	0,58	0,57	764
PPL parallel buffered sort	2,31	1,39	0,9	0,88	1,1	0,89	1,44	1527
Boost parallel sort	2,15	1,21	0,7	0,72	0,41	0,51	0,54	764
Boost sample sort	3,4	1,94	1,56	1,41	2	1,41	1,96	1526
Boost parallel stable sort	3,56	2,37	1,79	1,45	1,72	1,34	1,44	1145

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.