FLAT_STABLE_SORT

New stable sort algorithm

1.- INTRODUCTION

f**lat_stable_sort** is a new stable sort algorithm, created and implemented by the author, which use a very low additional memory (around 1% of the data size). The best case is O(N), and the average and worst case are O(NlogN).

The size of the additional memory is : size of the data / 256 + 8K

Data size	**Additional memory**	**Percent**
1M	12 K	1.2 %
1G	4M	0.4 %

The algorithm is fast sorting unsorted elements, but had been designed for to be extremely efficient when the data are near sorted. By example :

- Sorted elements with unsorted elements added at end or at he beginning .
- Sorted elements with unsorted elements inserted in internal positions, or elements modified which alter the ordered sequence.
- Reverse sorted elements
- Combination of the three previous points.

The results obtained in the sorting of 100 000 000 numbers plus a percent of unsorted elements inserted at end or in the middle , was

```
random                        |10.78 |
                              |      |
sorted                        | 0.07 |
sorted + 0.1% end             | 0.36 |
sorted +   1% end             | 0.49 |
sorted +  10% end             | 1.39 |
                              |      |
sorted + 0.1% middle          | 2.47 |
sorted +   1% middle          | 3.06 |
sorted +  10% middle          | 5.46 |
                              |      |
reverse sorted                | 0.14 |
reverse sorted + 0.1% end     | 0.41 |
reverse sorted +   1% end     | 0.55 |
reverse sorted +  10% end     | 1.46 |
                              |      |
reverse sorted + 0.1% middle| 2.46 |
reverse sorted +   1% middle| 3.16 |
reverse sorted +  10% middle| 5.46 |
```

The results obtained with strings and objects of several sizes with different comparisons, was equally satisfactory and according with the expected.

2.- DESCRIPTION OF THE ALGORITHM

2.1.- INTRODUCTION

The problem of the merge algorithms is where put the merged elements?. This is the justification of the additional memory of the stable sort algorithms (usually a half of the memory used by the data)

This algorithm have a different strategy. The data are grouped in blocks of fixed size. All elements inside a block are ordered. Suppose, initially, than the number of elements is a multiple of the block size. Further we see when this is not true.

Other important concept is the sequence. We can have several blocks ordered, but the blocks are not physically contiguous. We have a vector of positions, indicating the physically position of the blocks logically ordered by the vector of positions. The sequence is defined by an iterator to the first and other to the after the last element of the vector position.

By example

5	*6*	*7*	*8*
10	12	8	6

This sequence [5, 9), imply that the blocks of the sequence are in the physically positions 10, 12, 8 and 6. The data inside this sequence of non contiguous blocks are ordered

The idea of the algorithms is similar to others merge algorithms. Initially sort N blocks and have N sequences of 1 block. Merge the sequence 0 and 1, 2 and 3, 3 and 4 and , at the end have N/2 sequences of two blocks. In the new sequences obtained , merge the 0 and 1, 2 and 3 … and obtain N/ 4 sequences.

At end of this process, we have only 1 sequence. The logical order is in a vector of positions, using this vector and with a simple algorithms, move the elements from their physically position to the logic position, and the process is finished. This process is done one time and is the last operation of the algorithm.

2.1.1.- Block merge

In order to merge the blocks, use a “mixer”. This is a circular buffer, very simple and very fast, with an internal size of the double of the block size. The internal memory of the mixer, is use in other moments by the algorithm

The idea is merge two blocks, until one of them is empty. The merged elements are inside the mixer

2.1.2.- Sequence merge

We have two sequences, defined as two ranges of positions in a vector . The ranges can be of the same or distinct vector. By example [0, 4) and [4, 8).

	0	*1*	*2*	*3*
Sequence 1	1	2	0	3

	4	*5*	*6*	*7*
Sequence 2	6	5	4	7

Begin to merge the block1 1 and 6. If the first empty block is the 1, fill the block 1 with the data from the front of the mixer, and add 1 to output list.

Now, continue with the remaining elements of the block 6 and the next block of the first sequence, in this example the 2. When any of the two blocks is empty, is filled with the front data of the mixer, and their position is inserted in the output list.

For to explain, when a sequence is empty, we do with an example.
Suppose the sequence 1 is empty

	0	*1*	*2*	*3*
Sequence 1				

	4	*5*	*6*	*7*
Sequence 2			4	7

In the mixer, we have merged elements, and the block 4 is partially filled. The elements needed for to fill the block 4 are in the mixer. We move from the mixer to the block 4 and in the output list we add 4 and 7.

We have a new sequence, which can be like this

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Output sequence	1	6	2	0	5	3	4	7

The elements in the blocks of a sequence are ordered.

2.1.3.- Example

We have 4 blocks of 4 elements each. Initially, sorted the blocks, and now all the elements of a blocks are sorted.

The vector of positions of the blocks is called index. For to merge two sequences, copy the sequences from the index to two vectors, and the output sequence is generated over the index.

	0	1	2	3
Index	0	1	2	3

	0	1	2	3
Data	5, 7, 12, 23	4, 6, 11, 15	1, 9, 13, 34	8, 14, 17,19

We have 4 sequences of 1 block. Merge the sequences 0 with 1 and 2 with 3. After this we will have two sequences of two blocks. When merge the two sequences of two blocks , we obtain a sequence of 4 blocks, which is the final sequence.

Merge of blocks 0 and 1

Mixer	*Block 0*	*Block 1*
4,5,6,7,11,12,15	23	

The block 1 is empty. We fill with the element from the front of the mixer. And add 1 to output sequence. The first sequence is empty. The remaining elements of the mixer are moved to the front of the block 0, and add the 0 to the output sequence. The output sequence is [0, 2). We see in the index

	0	1	2	3
Index	1	0	2	3

	0	1	2	3
Data	11, 12, 15, 23	4,5,6, 7	1, 9, 13, 34	8, 14, 17,19

Doing the same with the blocks 2 and 3, we obtain

	0	1	2	3
Index	1	0	3	2

	0	1	2	3
Data	11, 12, 15, 23	4, 5, 6, 7	14, 17, 19, 34	1, 8, 9, 13

We have now, two sequences of two blocks each. The sequences are defined in the index, [0, 2) and [2, 4). The first sequence are the positions 0 and 1 of the index, and the second are the positions 2 and 3.

For to merge, take the firs block of the first sequence (1), with the first block of the second sequence (3), and begin the merge.

The first empty block is the 1. Now we fill the block 1 with the front data of the mixer, and add 1 to the output sequence. For to substitute the block 1 , take the next of the first sequence, the block 0, and continue with the merge.

Now the first empty block is the 3. We fill from the front of the mixer, and insert the number in the output sequence. For to substitute the block 3, take the next block of the sequence, the 2, and continue with the merge.

The next empty block is the 0. We fill from the front of the mixer , and add their number to the output sequence. The first sequence is empty. Now we have only the block 2 partially filled, we fill from the mixer and add their position to the output list

At end, we can see

	0	1	2	3
Index	1	3	0	2

	0	1	2	3
Data	12, 13, 14, 15	1, 4, 5, 6	17, 19, 23, 34	7, 8, 9, 11

Now, we must move the blocks, with a very simple algorithm, and pass to logical sorted with a index to a physically order. This is done only one time and is the last operation of the algorithm.

	0	1	2	3
Index	0	1	2	3

	0	1	2	3
Data	1, 4, 5, 6	7, 8, 9, 11	12, 13, 14, 15	17, 19, 23, 34

2.1.4.- Internal details

The additional memory needed by the algorithm are the two blocks of the mixer, and the list with the positions of the blocks. When merge two sequences, copy each sequence in a vector , and the result is over the index. In the last merge, the sum of the size of these two vectors is the same than the index. Due this, need the size of the index, multiply by two.

In the merge process, participate two blocks, and the two blocks of the mixer. The size of the blocks is designed for to allocate the 4 blocks in the L1 cache of the processor. Normally they have 32K for core, and each core have two threads, then , we have 16K for each thread.

Due this the block size must be 4K, and then: Size of one object x Block size = 4K

Size of the object	**Block Size**
4	1024
8	512
16	256
32	128
64	64

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

When the number of elements to sort is not a multiple of the block size, we have an incomplete final block, called tail, and always is the last. When merge two sequences, if we have tail, always is in the second sequence.

When exist tail block in the second sequence, the merge is as showed before. If the first sequence is empty, the procedure is as described before.

But if the sequence empty is the second, and have tail, must do a different procedure. We see with an example

	0	1	2	3	4	5
Index	1	2	0	4	3	5

	0	1	2	3	4	5
Data	38, 42, 44, 46	1, 5, 13, 19	23, 29, 34, 36	22, 25, 27, 28	4, 14, 17, 20	31, 32

We want to merge the sequence [0, 3), y [3, 6). The block 5 is incomplete or tail block.

Begin the merge, and the second sequence is empty. In this instant the state of the data is

	0	1	2
Sequence 1		2	0

The block 2 is partially empty.

	0	1	2
Sequence 2			

	0	1	2	3	4	5
Output sequence	1	4	3			

	0	1	2	3	4	5
Data	38, 42, 44, 46	1, 4, 5, 13	34, 36	22, 23, 25, 27	14, 17, 19, 20	

Mixer	28, 29, 31, 32

Now, we add the tail block to the end of the first sequence.

	2	0	5
Data pending of the sequence 1	34, 36	38, 42, 44, 46	

The tail block have a size of two in this example. We shift to the right the data pending, the size of the tail block (2 positions), and insert the data of the mixer by the left side, and we have

	2	0	5
Data pending of the sequence 1	28, 29, 31, 32	34, 36, 38, 42	44, 46

Now, we have the blocks filled, and now we must insert their numbers in the output sequence

	0	1	2	3	4	5
Output sequence	1	4	3	2	0	5

	0	1	2	3	4	5
Data	34, 36, 38, 42	1, 4, 5, 13	28, 29, 31, 32	22, 23, 25, 27	14, 17, 19, 20	44, 46

Now, only must move the blocks from the physical position to their logical position, and all the data are sorted.

2.3.- SPECIAL CASES

The algorithm had been designed for to be extremely efficient with the data near sorted. We can classify in four cases:

- Sorted elements, and add unsorted elements to the beginning or the end.
- Sorted elements and unsorted elements are inserted in internal positions, or elements modified, which alter the order of the elements.
- Reverse sorted elements
- Combination of the first 3 cases

Begin from the first position, looking for sorted or reverse sorted elements. If the number of sorted or reverse sorted elements is greater than a value (usually ¼ of the number of elements), begin with an special process. If not, apply the general process described previously.

For to explain the special process, we do with an example

If the elements are reverse sorted, move between them for to be sorted.
By example, if have an array of 16 elements, with a block size of 4

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
3	4	5	7	8	10	13	14	17	21	6	12	9	11	2	16

Range 1, 10 sorted elements *Range 2, 6 unsorted elements*

Sort the range 2, and obtain

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
3	4	5	7	8	10	13	14	17	21	2	6	9	11	12	16

Range 1, 10 sorted elements *Range 2, 6 sorted elements*

The idea is to merge the two ranges, and can appear two cases:

1. When the range 2 is lower or equal than the double of the block size
2. When the range 2 is greater than the double of the block size

2.3.1.- Number of elements lower or equal than the double of the block size.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
3	4	5	7	8	10	13	14	17	21	6	12	9	11	2	16

Sort the range 2, and after this, the idea is to use as auxiliary memory the internal memory of the mixed (2 blocks). And do an insertion of sorted elements of the range 2 in the sorted elements of the range 1

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
3	4	5	7	8	10	13	14	17	21	2	6	9	11	12	16

Move the range 2 in the auxiliary memory and obtain

0	*1*	*2*	*3*	*4*	*5*
2	6	9	11	12	16

Find the position of insertion of each element, and calculate the positions to shift the elements

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	4	5	7	8	10	13	14	17	21						

1 position *2 positions* *3 pois-tions* *5 positions* *6 positions*

Shift the elements and insert the elements in the auxiliary memory and obtain

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	16	17	21

And now, we have all the elements sorted

2.3.2.- Number of elements greater than the double of the block size.

In the description of the algorithm, we have a merge of two sequences, but have a limitation. In the first sequence, all the blocks must be completed, and the number of elements multiple of the block size. The second sequence, don't have this limitation, because can have a tail block..

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	4	5	7	8	10	13	14	17	21	6	12	9	11	2	16

Range 1, 10 elements sorted *Range 2, 6 elements unsorted*

If we have a block size of 4, and the first range have 10 elements, we cut the number of elements of this range to the nearest multiple of the block size, lower than the actual size. In this case 8, and after this the first range have 8 elements, and the second range 8 elements to sort.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	4	5	7	8	10	13	14	17	21	6	12	9	11	2	16

New Range 1 8 sorted elements *New Range 2 8 unsorted elements*

Sort the range 2 and obtain

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	4	5	7	8	10	13	14	2	6	9	11	12	16	17	21

New Range 1 of 8 sorted elements *New Range 2 of 8 sorted elements*

Now, the range 1 have a number of elements multiple of the block size, and we can do the merge as described in the general case in the point 2.1.

2.3.3.- Look for sorted elements from the end to the beginning

In the same way that we begin to look for sorted or reverse sorted elements from the beginning to the end, we will do from the end to the beginning. The ideas and concepts are identical than in the look for forward

If the number of elements sorted or reverse sorted is greater or equal than ¼ of the number of elements, apply an special process. If the number of elements pending to sort is lower or equal than the double of the block size, we do an insertion similar to the previously described, and if it is greater, cut the number of elements of the range 2 , for to obtain a range 1 with a number of elements multiple of the block size, the sort, and after do the merge with the general process.

3.- BENCHMARKS

The measured memory in the sorting of 100 000 000 numbers of 64 bits was:

```
std::stable_sort        1176 MB
Boost spin_sort         1175 MB
Boost flat_stable_sort   787 MB
```

In the time benchmark, the random, sorted and reverse sorted elements are 100000000 numbers of 64 bits. To these numbers, add 0.1%, 1% and 10% of unsorted elements inserted at the end and in the middle, uniformly spaced.

[1] std::stable_sort [2] Boost spin_sort [3] Boost flat_stable_sort

	[1]	[2]	[3]
random	8.51	9.45	10.78
sorted	4.86	0.06	0.07
sorted + 0.1% end	4.89	0.41	0.36
sorted + 1% end	4.96	0.55	0.49
sorted + 10% end	5.71	1.31	1.39
sorted + 0.1% middle	6.51	1.85	2.47
sorted + 1% middle	7.03	2.07	3.06
sorted + 10% middle	9.42	3.92	5.46
reverse sorted	5.10	0.13	0.14
reverse sorted + 0.1% end	5.21	0.52	0.41
reverse sorted + 1% end	5.27	0.65	0.55
reverse sorted + 10% end	6.01	1.43	1.46
reverse sorted + 0.1% middle	6.51	1.85	2.46
reverse sorted + 1% middle	7.03	2.07	3.16
reverse sorted + 10% middle	9.42	3.92	5.46