Boost.Sort

Steven Ross

Table of Contents

Overview

Introduction

The Boost.Sort library provides a generic implementation of high-speed sorting algorithms that outperform those in the C++ standard in both average and worst case performance when there are over 1000 elements in the list to sort.

They fall back to STL std::sort on small data sets.

> **Warning**
>
> These algorithms all only work on random access iterators.

They are hybrids using both radix and comparison-based sorting, specialized to sorting common data types, such as integers, floats, and strings.

These algorithms are encoded in a generic fashion and accept functors, enabling them to sort any object that can be processed like these basic data types. In the case of `string_sort`, this includes anything with a defined strict-weak-ordering that std::sort can sort, but writing efficient functors for some complex key types may not be worth the additional effort relative to just using std::sort, depending on how important speed is to your application. Sample usages are available in the example directory.

Unlike many radix-based algorithms, the underlying spreadsort algorithm is designed around **worst-case performance**. It performs better on chunky data (where it is not widely distributed), so that on real data it can perform substantially better than on random data. Conceptually, spreadsort can sort any data for which an absolute ordering can be determined, and `string_sort` is sufficiently flexible that this should be possible.

Situations where spreadsort is fastest relative to std::sort:

1. Large number of elements to sort (*N* >= 10000).
2. Slow comparison function (such as floating-point numbers on x86 processors or strings).
3. Large data elements (such as key + data sorted on a key).
4. Completely sorted data when spreadsort has an optimization to quit early in this case.

Situations where spreadsort is slower than std::sort:

1. Data sorted in reverse order. Both std::sort and spreadsort are faster on reverse-ordered data than randomized data, but std::sort speeds up more in this special case.
2. Very small amounts of data (< 1000 elements). For this reason there is a fallback in spreadsort to std::sort if the input size is less than 1000, so performance is identical for small amounts of data in practice.

These functions are defined in `namespace boost::sort::spreadsort`.

Overloading

> **Tip**
>
> In the Boost.Sort C++ Reference section, click on the appropriate overload, for example `float_sort(RandomAccessIter, RandomAccessIter, Right_shift, Compare);` to get full details of that overload.

Each of `integer_sort`, `float_sort`, and `string_sort` have 3 main versions: The base version, which takes a first iterator and a last iterator, just like std::sort:

```
integer_sort(array.begin(), array.end());
```

The version with an overridden shift functor, providing flexibility in case the `operator>>` already does something other than a bitshift. The rightshift functor takes two args, first the data type, and second a natural number of bits to shift right.

For `string_sort` this variant is slightly different; it needs a bracket functor equivalent to `operator`[], taking a number corresponding to the character offset, along with a second `getlength` functor to get the length of the string in characters. In all cases, this operator must return an integer type that compares with the `operator<` to provide the intended order (integers can be negated to reverse their order).

In other words:

```
rightshift(A, n) < rightshift(B, n) -> A < B
```

```
integer_sort(array.begin(), array.end(), rightshift());
```

See rightshiftsample.cpp for a worked example.

And a version with a comparison functor for maximum flexibility. This functor must provide the same sorting order as the integers returned by the rightshift:

```
rightshift(A, n) < rightshift(B, n) -> compare(A, B)
```

```
integer_sort(array.begin(), array.end(), negrightshift(), std::greater<DATA_TYPE>());
```

Examples of functors are:

```
struct lessthan {
  inline bool operator()(const DATA_TYPE &x, const DATA_TYPE &y) const {
    return x.a < y.a;
  }
};
```

```
struct bracket {
  inline unsigned char operator()(const DATA_TYPE &x, size_t offset) const {
    return x.a[offset];
  }
};
```

```
struct getsize {
  inline size_t operator()(const DATA_TYPE &x) const{ return x.a.size(); }
};
```

and this is used thus:

```
string_sort(array.begin(), array.end(), bracket(), getsize(), lessthan());
```

See stringfunctorsample.cpp for a worked example of a string with functor example.

TODO I find this above confusing (and may be confused - haven't stopped to think carefully) - I think you need links to all the examples and use snippets here.

Performance

The spreadsort algorithm is a hybrid algorithm; when the number of elements being sorted is below a certain number, comparison-based sorting is used. Above it, radix sorting is used. The radix-based algorithm will thus cut up the problem into small pieces, and either completely sort the data based upon its radix if the data is clustered, or finish sorting the cut-down pieces with comparison-based sorting.

The Spreadsort algorithm dynamically chooses either comparison-based or radix-based sorting when recursing, whichever provides better worst-case performance. This way worst-case performance is guaranteed to be the better of $(N \cdot log2(N))$ comparisons and $(N \cdot log2(K/S + S))$ operations where

- N is the number of elements being sorted,
- K is the length in bits of the key, and
- S is a constant.

This results in substantially improved performance for large N; `integer_sort` tends to be 50% to 2X faster than std::sort, while `float_sort` and _string_sort are roughly 2X faster than std::sort.

Performance graphs are provided for `integer_sort`, `float_sort`, and `string_sort` in their description.

Runtime Performance comparisons and graphs were made on a Core 2 Duo laptop running Windows Vista 64 with MSVC 8.0, and an old G4 laptop running Mac OSX with gcc. Boost bjam/b2 was used to control compilation.

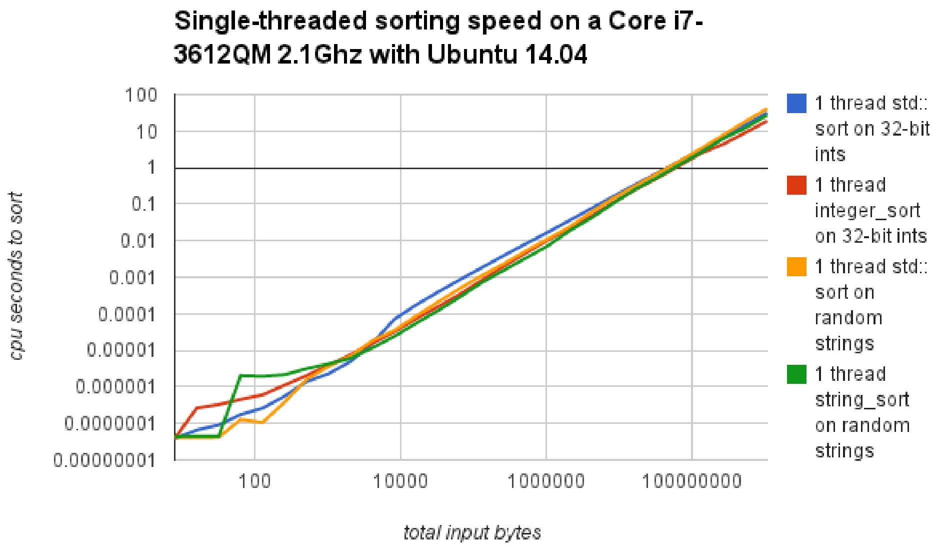
Direct performance comparisons on a newer x86 system running Ubuntu, with the fallback to std::sort at lower input sizes disabled are below.

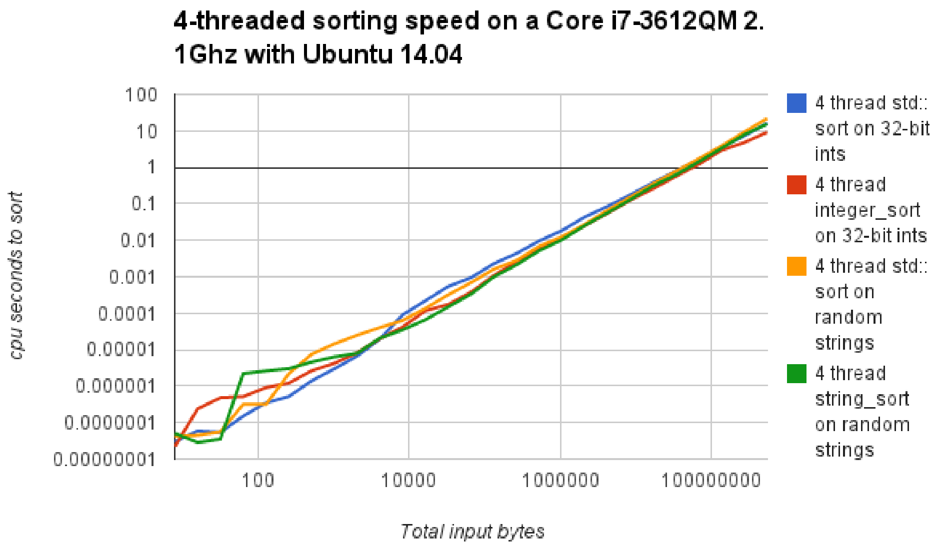
> **Note**
>
> The fallback to std::sort for smaller input sizes prevents the worse performance seen on the left sides of the first two graphs.

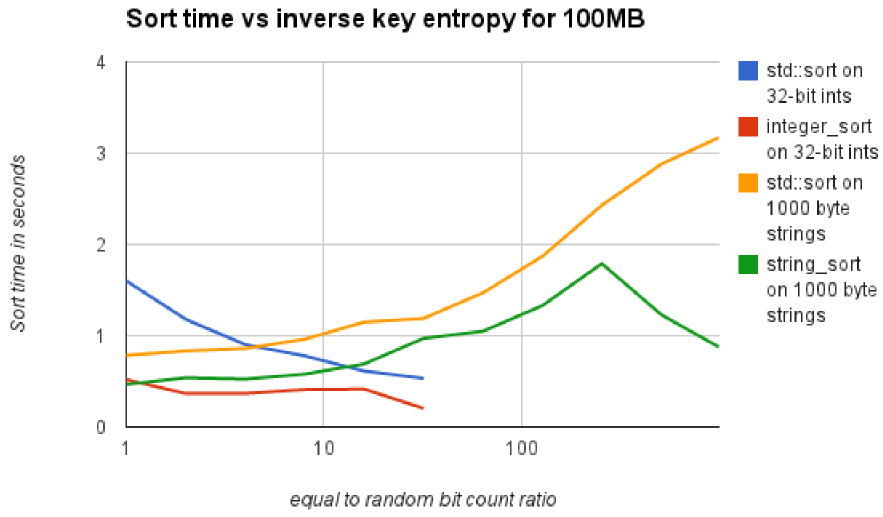
`integer_sort` starts to become faster than std::sort at about 1000 integers (4000 bytes), and `string_sort` becomes faster than std::sort at slightly fewer bytes (as few as 30 strings).

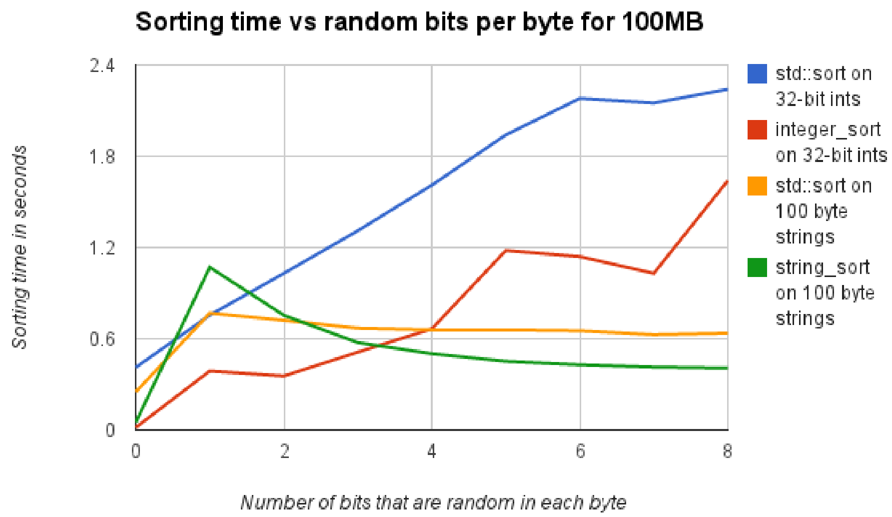
> **Note**
>
> The 4-threaded graph has 4 threads doing **separate sorts simultaneously** (not splitting up a single sort) as a test for thread cache collision and other multi-threaded performance issues.

`float_sort` times are very similar to `integer_sort` times.


Single-threaded sorting speed on a Core i7-3612QM 2.1Ghz with Ubuntu 14.04
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0.000001
0.0000001
0.00000001
cpu seconds to sort
100
10000
1000000
100000000
total input bytes
1 thread std::sort on 32-bit ints
1 thread integer_sort on 32-bit ints
1 thread std::sort on random strings
1 thread string_sort on random strings


4-threaded sorting speed on a Core i7-3612QM 2.1Ghz with Ubuntu 14.04
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0.000001
0.0000001
0.00000001
cpu seconds to sort
100
10000
1000000
100000000
Total input bytes
4 thread std::sort on 32-bit ints
4 thread integer_sort on 32-bit ints
4 thread std::sort on random strings
4 thread string_sort on random strings





Histogramming with a fixed maximum number of splits is used because it reduces the number of cache misses, thus improving performance relative to the approach described in detail in the original SpreadSort publication.

The importance of cache-friendly histogramming is described in Arne Maus, Adaptive Left Reflex, though without the worst-case handling described below.

The time taken per radix iteration is:

(N) iterations over the data

(N) integer-type comparisons (even for _float_sort and `string_sort`)

(N) swaps

(2^S) bin operations.

To obtain *(N)* worst-case performance per iteration, the restriction $S <= log2(N)$ is applied, and *(2^S)* becomes *(N)*. For each such iteration, the number of unsorted bits log2(range) (referred to as *K*) per element is reduced by *S*. As *S* decreases depending upon the amount of elements being sorted, it can drop from a maximum of S_{max} to the minimum of S_{min}.

Assumption: std::sort is assumed to be $(N*log2(N))$, as introsort exists and is commonly used. (If you have a quibble with this please take it up with the implementor of your std::sort; you're welcome to replace the recursive calls to std::sort to calls to introsort if your std::sort library call is poorly implemented).

Introsort is not included with this algorithm for simplicity and because the implementor of the std::sort call is assumed to know what they're doing.

To maintain a minimum value for S (S_{min}), comparison-based sorting has to be used to sort when $n <= log2(meanbinsize)$, where $log2(meanbinsize)$ (lbs) is a small constant, usually between 0 and 4, used to minimize bin overhead per element. There is a small corner-case where if $K < S_{min}$ and $n >= 2$^K, then the data can be sorted in a single radix-based iteration with an $S = K$ (this bucketsorting special case is by default only applied to `float_sort`). So for the final recursion, worst-case performance is:

1 radix-based iteration if $K <= S_{min}$,

or $S_{min} + lbs$ comparison-based iterations if $K > S_{min}$ but $n <= 2^{(S_{min} + lbs)}$.

So for the final iteration, worst-case runtime is $(N*(S_{min} + lbs))$ but if $K > S_{min}$ and $N > 2^{(S_{min} + lbs)}$ then more than 1 radix recursion will be required.

For the second to last iteration, $K <= S_{min} * 2 + 1$ can be handled, (if the data is divided into $2^{(S_{min} + 1)}$ pieces) or if $N < 2^{(S_{min} + lbs + 1)}$, then it is faster to fallback to std::sort.

In the case of a radix-based sort plus recursion, it will take $(N*(S_{min} + lbs)) + (N) = (N*(S_{min} + lbs + 1))$ worst-case time, as $K\_remaining = K\_start - (S_{min} + 1)$, and $K\_start <= S_{min} * 2 + 1$.

Alternatively, comparison-based sorting is used if $N < 2^{(S_{min} + lbs + 1)}$, which will take $(N*(S_{min} + lbs + 1))$ time.

So either way $(N*(S_{min} + lbs + 1))$ is the worst-case time for the second to last iteration, which occurs if $K <= S_{min} * 2 + 1$ or $N < 2^{(S_{min} + lbs + 1)}$.

This continues as long as $S_{min} <= S <= S_{max}$, so that for $K\_m <= K\_(m-1) + S_{min} + m$ where m is the maximum number of iterations after this one has finished, or where $N < 2^{(S_{min} + lbs + m)}$, then the worst-case runtime is $(N*(S_{min} + lbs + m))$.

K_m at $m <= (S_{max} - S_{min})$ works out to:

$K\_1 <= (S_{min}) + S_{min} + 1 <= 2S_{min} + 1$

$K\_2 <= (2S_{min} + 1) + S_{min} + 2$

as the sum from 0 to m is $m(m + 1)/2$

$K\_m <= (m + 1)S_{min} + m(m + 1)/2 <= (S_{min} + m/2)(m + 1)$

substituting in $S_{max} - S_{min}$ for m

$K\_(S_{max} - S_{min}) <= (S_{min} + (S_{max} - S_{min})/2)*(S_{max} - S_{min} + 1)$

$K\_(S_{max} - S_{min}) <= (S_{min} + S_{max}) * (S_{max} - S_{min} + 1)/2$

Since this involves $S_{max} - S_{min} + 1$ iterations, this works out to dividing K into an average $(S_{min} + S_{max})/2$ pieces per iteration.

To finish the problem from this point takes $(N * (S_{max} - S_{min}))$ for m iterations, plus the worst-case of $(N*(S_{min} + lbs))$ for the last iteration, for a total of $(N *(S_{max} + lbs))$ time.

When $m > S_{max} - S_{min}$, the problem is divided into S_{max} pieces per iteration, or std::sort is called if $N < 2$^$(m + S_{min} + lbs)$. For this range:

$K\_m <= K\_(m - 1) + S_{max}$, providing runtime of

$(N *((K - K\_(S_{max} - S_{min}))/S_{max} + S_{max} + lbs))$ if recursive,

or $(N * log(2\^(m + S_{min} + lbs)))$ if comparison-based,

which simplifies to $(N * (m + S_{min} + lbs))$, which substitutes to $(N * ((m - (S_{max} - S_{min})) + S_{max} + lbs))$, which given that $m - (S_{max} - S_{min}) <= (K - K\_(S_{max} - S_{min}))/S_{max}$ (otherwise a lesser number of radix-based iterations would be used)

also comes out to $(N *((K - K\_(S_{max} - S_{min}))/S_{max} + S_{max} + lbs))$.

Asymptotically, for large N and large K, this simplifies to:

$(N * (K/S_{max} + S_{max} + lbs))$,

simplifying out the constants related to the $S_{max} - S_{min}$ range, providing an additional $(N * (S_{max} + lbs))$ runtime on top of the $(N * (K/S))$ performance of LSD radix sort, but without the (N) memory overhead. For simplicity, because *lbs* is a small constant (0 can be used, and performs reasonably), it is ignored when summarizing the performance in further discussions. By checking whether comparison-based sorting is better, Spreadsort is also $(N*log(N))$, whichever is better, and unlike LSD radix sort, can perform much better than the worst-case if the data is either evenly distributed or highly clustered.

This analysis was for `integer_sort` and `float_sort`. `string_sort` differs in that $S_{min} = S_{max} = sizeof(Char\_type) * 8$, *lbs* is 0, and that std::sort's comparison is not a constant-time operation, so strictly speaking `string_sort` runtime is

$(N * (K/S_{max} + (S_{max}\ comparisons)))$.

Worst-case, this ends up being $(N * K)$ (where K is the mean string length in bytes), as described for American flag sort, which is better than the

$(N * K * log(N))$

worst-case for comparison-based sorting.

Tuning

`integer_sort` and `float_sort` have tuning constants that control how the radix-sorting portion of those algorithms work. The ideal constant values for `integer_sort` and `float_sort` vary depending on the platform, compiler, and data being sorted. By far the most important constant is *max_splits*, which defines how many pieces the radix-sorting portion splits the data into per iteration.

The ideal value of *max_splits* depends upon the size of the L1 processor cache, and is between 10 and 13 on many systems. A default value of 11 is used. For mostly-sorted data, a much larger value is better, as swaps (and thus cache misses) are rare, but this hurts runtime severely for unsorted data, so is not recommended.

On some x86 systems, when the total number of elements being sorted is small (less than 1 million or so), the ideal *max_splits* can be substantially larger, such as 17. This is suspected to be because all the data fits into the L2 cache, and misses from L1 cache to L2 cache do not impact performance as severely as misses to main memory. Modifying tuning constants other than *max_splits* is not recommended, as the performance improvement for changing other constants is usually minor.

If you can afford to let it run for a day, and have at least 1GB of free memory, the perl command: `./tune.pl -large -tune` (UNIX) or `perl tune.pl -large -tune -windows` (Windows) can be used to automatically tune these constants. This should be run from the `libs/sort/sort directory` inside the boost home directory. This will work to identify the `ideal constants.hpp` settings for your system, testing on various distributions in a 20 million element (80MB) file, and additionally verifies that all sorting routines sort correctly across various data distributions. Alternatively, you can test with the file size you're most concerned with `./tune.pl number -tune` (UNIX) or `perl tune.pl number -tune -windows` (Windows). Substitute the number of elements you want to test with for `number`. Otherwise, just use the options it comes with, they're decent. With default settings `./tune.pl -tune` (UNIX) `perl tune.pl -tune -windows` (Windows), the script will take hours to run (less than a day), but may not pick the correct *max_splits* if it is over 10. Alternatively, you can add the `-small` option to make it take just a few minutes, tuning for smaller vector sizes (one hundred thousand elements), but the resulting constants may not be good for large files (see above note about *max_splits* on Windows).

The tuning script can also be used just to verify that sorting works correctly on your system, and see how much of a speedup it gets, by omiting the "-tune" option. This runs at the end of tuning runs. Default args will take about an hour to run and give accurate results on decent-sized test vectors. `./tune.pl -small` (UNIX) `perl tune.pl -small -windows` (Windows) is a faster option, that tests on smaller vectors and isn't as accurate.

If any differences are encountered during tuning, please call `tune.pl` with `-debug > log_file_name`. If the resulting log file contains compilation or permissions issues, it is likely an issue with your setup. If some other type of error is encountered (or result differences), please send them to the library author at spreadsort@gmail.com. Including the zipped `input.txt` that was being used is also helpful.

Spreadsort

Header <boost/sort/spreadsort/spreadsort.hpp>

`spreadsort` checks whether the data-type provided is an integer, castable float, string, or wstring.

- If data-type is an integer, `integer_sort` is used.
- If data-type is a float, `float_sort` is used.
- If data-type is a string or wstring, `string_sort` is used.
- Sorting other data-types requires picking between `integer_sort`, `float_sort` and `string_sort` directly, as spreadsort won't accept types that don't have the appropriate type traits.

Overloading variants are provided that permit use of user-defined right-shift functors and comparison functors.

Each function is optimized for its set of arguments; default functors are not provided to avoid the risk of any reduction of performance.

See overloading section.

Rationale:

`spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data-type, enabling faster generic programming.

Spreadsort Examples

See example folder for all examples.

See sample.cpp for a simple worked example.

For an example of 64-bit integer sorting, see int64.cpp.

This example sets the element type of a vector to 64-bit integer

```
#define DATA_TYPE boost::int64_t
```

and calls the sort

```
boost::sort::spreadsort(array.begin(), array.end());
```

See rightshiftsample.cpp for a worked example of using rightshift, using a user-defined functor:

```
struct rightshift {
  inline int operator()(DATA_TYPE x, unsigned offset) { return x >> offset; }
};
```

For a simple example sorting `floats`,

```
vector<float> vec;
vec.push_back(1.0);
vec.push_back(2.3);
vec.push_back(1.3);
...
spreadsort(vec.begin(), vec.end());
//The sorted vector contains "1.0 1.3 2.3 ..."
```

See also floatsample.cpp which checks for abnormal values.

While floatfunctorsample.cpp shows use of sorting float-point types with functors.

```
#define CAST_TYPE int
#define KEY_TYPE float
```

```
struct DATA_TYPE {
    KEY_TYPE key;
    std::string data;
};
```

Right-shift functor:

```
// Casting to an integer before bitshifting
struct rightshift {
  int operator()(const DATA_TYPE &x, const unsigned offset) const {
    return float_mem_cast<KEY_TYPE, CAST_TYPE>(x.key) >> offset;
  }
};
```

Comparison lessthan `operator<` functor:

```
struct lessthan {
  bool operator()(const DATA_TYPE &x, const DATA_TYPE &y) const {
    return x.key < y.key;
  }
};
```

Integer Spreadsort

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than std::sort for large tests (>=100kB). Worst-case performance is *(N * (log2(range)/s + s))*, so `integer_sort` is asymptotically faster than pure comparison-based algorithms. *s* is *max_splits*, which defaults to 11, so its worst-case with default settings for 32-bit integers is *(N * ((32/11)* slow radix-based iterations + 11 fast comparison-based iterations).

Some performance plots of runtime vs. n and log2(range) are provided below:

Windows Integer Sort

OSX integer Sort

Integer Sort Examples

Key plus data sample. ... TODO

Rationale

Radix Sorting

Radix-based sorting allows the data to be divided up into more than 2 pieces per iteration, and for cache-friendly versions, it normally cuts the data up into around a thousand pieces per iteration. This allows many fewer iterations to be used to complete sorting the data, enabling performance superior to the *(N*log(N))* comparison-based sorting limit.

Hybrid Radix

There a two primary types of radix-based sorting:

Most-significant-digit Radix sorting (MSD) divides the data recursively based upon the top-most unsorted bits. This approach is efficient for even distributions that divide nicely, and can be done in-place (limited additional memory used). There is substantial constant overhead for each iteration to deal with the splitting structure. The algorithms provided here use MSD Radix Sort for their radix-sorting portion. The main disadvantage of MSD Radix sorting is that when the data is cut up into small pieces, the overhead for additional recursive calls starts to dominate runtime, and this makes worst-case behavior substantially worse than *(N*log(N))*.

By contrast, `integer_sort`, `float_sort`, and `string_sort` all check to see whether Radix-based or Comparison-based sorting have better worst-case runtime, and make the appropriate recursive call. Because Comparison-based sorting algorithms are efficient on small pieces, the tendency of MSD radix sort to cut the problem up small is turned into an advantage by these hybrid sorts. It is hard to conceive of a common usage case where pure MSD radix sort would have any significant advantage over hybrid algorithms.

Least-significant-digit radix sort (LSD) sorts based upon the least-significant bits first. This requires a complete copy of the data being sorted, using substantial additional memory. The main advantage of LSD Radix Sort is that aside from some constant overhead and the memory allocation, it uses a fixed amount of time per element to sort, regardless of distribution or size of the list. This amount of time is proportional to the length of the radix. The other advantage of LSD Radix Sort is that it is a stable sorting algorithm, so elements with the same key will retain their original order.

One disadvantage is that LSD Radix Sort uses the same amount of time to sort "easy" sorting problems as "hard" sorting problems, and this time spent may end up being greater than an efficient *(N*log(N))* algorithm such as introsort spends sorting "hard" problems on large data sets, depending on the length of the datatype, and relative speed of comparisons, memory allocation, and random accesses.

The other main disadvantage of LSD Radix Sort is its memory overhead. It's only faster for large data sets, but large data sets use significant memory, which LSD Radix Sort needs to duplicate. LSD Radix Sort doesn't make sense for items of variable length, such as strings; it could be implemented by starting at the end of the longest element, but would be extremely inefficient.

All that said, there are places where LSD Radix Sort is the appropriate and fastest solution, so it would be appropriate to create a templated LSD Radix Sort similar to `integer_sort` and `float_sort`. This would be most appropriate in cases where comparisons are extremely slow.

Why spreadsort?

The spreadsort algorithm used in this library is designed to provide best possible worst-case performance, while still being cache-friendly. It provides the better of *(N*log(K/S + S))* and *(N*log(N))* worst-case time, where *K* is the log of the range. The log of the range is normally the length in bits of the data type; 32 for a 32-bit integer.

`flash_sort` (another hybrid algorithm), by comparison is *(N)* for evenly distributed lists. The problem is, `flash_sort` is merely an MSD radix sort combined with *(N*N)* insertion sort to deal with small subsets where the MSD Radix Sort is inefficient, so it is inefficient with chunks of data around the size at which it switches to `insertion_sort`, and ends up operating as an enhanced MSD Radix Sort. For uneven distributions this makes it especially inefficient.

`integer_sort` and `float_sort` use introsort instead, which provides *(N*log(N))* performance for these medium-sized pieces. Also, `flash_sort`'s *(N)* performance for even distributions comes at the cost of cache misses, which on modern architectures are extremely expensive, and in testing on modern systems ends up being slower than cutting up the data in multiple, cache-friendly steps. Also worth noting is that on most modern computers, `log(available RAM)/log2(L1 cache size)` is around 3, where a cache miss takes more than 3 times as long as an in-cache random-access, and the size of *max_splits* is tuned to the size of the cache. On a computer where cache misses aren't this expensive, *max_splits* could be increased to a large value, or eliminated entirely, and `integer_sort/float_sort` would have the same *(N)* performance on even distributions.

Adaptive Left Radix (ALR) is similar to `flash_sort`, but more cache-friendly. It still uses insertion_sort. Because ALR uses *(N*N)* `insertion_sort`, it isn't efficient to use the comparison-based fallback sort on large lists, and if the data is clustered in small chunks just over the fallback size with a few outliers, radix-based sorting iterates many times doing little sorting with high overhead. Asymptotically, ALR is still *(N*log(K/S + S))*, but with a very small *S* (about 2 in the worst case), which compares unfavorably with the 11 default value of *max_splits* for Spreadsort.

ALR also does not have the *(N*log(N))* fallback, so for small lists that are not evenly distributed it is extremely inefficient. See the `alrbreaker` and `binaryalrbreaker` testcases for examples; either replace the call to sort with a call to ALR and update the ALR_THRESHOLD at the top, or as a quick comparison make `get_max_count return ALR_THRESHOLD` (20 by default based upon the paper). These small tests take 4-10 times as long with ALR as std::sort in the author's testing, depending on the test system, because they are trying to sort a highly uneven distribution. Normal Spreadsort does much better with them, because `get_max_count` is designed around minimizing worst-case runtime.

`burst_sort` is an efficient hybrid algorithm for strings that uses substantial additional memory.

`string_sort` uses minimal additional memory by comparison. Speed comparisons between the two haven't been made, but the better memory efficiency makes `string_sort` more general.

`postal_sort` and `string_sort` are similar. A direct performance comparison would be welcome, but an efficient version of `postal_sort` was not found in a search for source.

`string_sort` is most similar to the American flag sort algorithm. The main difference is that it doesn't bother trying to optimize how empty buckets/piles are handled, instead just checking to see if all characters at the current index are equal. Other differences are using std::sort as the fallback algorithm, and a larger fallback size (256 vs. 16), which makes empty pile handling less important.

Another difference is not applying the stack-size restriction. Because of the equality check in `string_sort`, it would take *m*m* memory worth of strings to force `string_sort` to create a stack of depth *m*. This problem isn't a realistic one on modern systems with multi-megabyte stacksize limits, where main memory would be exhausted holding the long strings necessary to exceed the stacksize limit. `string_sort` can be thought of as modernizing American flag sort to take advantage of introsort as a fallback algorithm. In the author's testing, American flag sort (on `std::strings`) had comparable runtime to introsort, but making a hybrid of the two allows reduced overhead and substantially superior performance.

Unstable Sorting

Making a radix sort stable requires the usage of an external copy of the data. A stable hybrid algorithm also requires a stable comparison-based algorithm, and these are generally slow. LSD radix sort uses an external copy of the data, and provides stability, along with likely being faster (than a stable hybrid sort), so that's probably a better way to go for integer and floating-point types. It might make sense to make a stable version of `string_sort` using external memory, but for simplicity this has been left out for now.

Unused X86 optimization

Though the ideal *max_splits* for `n < 1 million` (or so) on x86 *seems* to be substantially larger, enabling a roughly 15% speedup for such tests, this optimization isn't general, and doesn't apply for `n > 1 million`. A too large *max_splits* can cause sort to take more than twice as long, so it should be set on the low end of the reasonable range, where it is right now.

Lookup Table?

The ideal way to optimize the constants would be to have a carefully-tuned lookup-table instead of the `get_max_count` function, but 4 tuning variables is simpler, `get_max_count` enforces worst-case performance minimization rules, and such a lookup table would be difficult to optimize for cross-platform performance.

Alternatively, `get_max_count` could be used to generate a static lookup table. This hasn't been done due to concerns about cross-platform compatibility and flexibility.

Definitions

stable sort

A sorting approach that preserves pre-existing order. If there are two elements with identical keys in a list that is later stably sorted, whichever came first in the initial list will come first in a stably sorted list. The algorithms provided here provide no such guarantee; items with identical keys will have arbitrary resulting order relative to each other.

Frequently asked Questions

There are no FAQs yet.

Acknowledgements

- The author would like to thank his wife Mary for her patience and support during the long process of converting this from a piece of C code to a template library.
- The author would also like to thank Phil Endecott and Frank Gennari for the improvements they've suggested and for testing. Without them this would have taken longer to develop or performed worse.
- `float_mem_cast` was fixed to be safe and fast thanks to Scott McMurray. That fix was critical for a high-performance cross-platform `float_sort`.
- Thanks also for multiple helpful suggestions provided by Steven Watanabe, Edouard Alligand, and others.
- Initial documentation was refactored to use Quickbook by Paul A. Bristow.

Bibliography

Standard Template Library Sort Algorithms

C++ STL sort algorithms.

Radix Sort

A type of algorithm that sorts based upon distribution instead of by comparison. Wikipedia has an article about Radix Sorting. A more detailed description of various Radix Sorting algorithms is provided here:

Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 5.2.5: Sorting by Distribution, pp.168-179.

Introsort

A high-speed comparison-based sorting algorithm that takes *(N * log(N))* time. See introsort and Musser, David R. (1997). "Introspective Sorting and Selection Algorithms", Software: Practice and Experience (Wiley) 27 (8), pp 983-993, available at Musser Introsoft.

America Flag Sort

A high-speed hybrid string sorting algorithm that `string_sort` is partially based upon. See American flag sort and Peter M. McIlroy, Keith Bostic, M. Douglas McIlroy. Engineering Radix Sort, Computing Systems 1993 at Engineering Radix sort, Peter M McIlroy and Keith Bostic.

Adaptive Left Radix (ARL)

ARL (Adaptive Left Radix) is a hybrid cache-friendly integer sorting algorithm with comparable speed on random data to `integer_sort`, but does not have the optimizations for worst-case performance, causing it to perform poorly on certain types of unevenly distributed data.

Arne Maus, ARL, a faster in-place, cache friendly sorting algorithm, presented at NIK2002, Norwegian Informatics Conference, Kongsberg, 2002. Tapir, ISBN 82-91116-45-8.

Original Spreadsort

The algorithm that `integer_sort` was originally based on. `integer_sort` uses a smaller number of key bits at a time for better cache efficiency than the method described in the paper. The importance of cache efficiency grew as CPU clock speeds increased while main memory latency stagnated. See Steven J. Ross, The Spreadsort High-performance General-case Sorting Algorithm, Parallel and Distributed Processing Techniques and Applications, Volume 3, pp.1100-1106. Las Vegas Nevada. 2002. See Steven Ross spreadsort_2002.

History

- First release following review in Boost 1.58.
- Review of Boost.Sort/Spreadsort library

Boost.Sort C++ Reference

Header <boost/sort/spreadsort/float_sort.hpp>

```
namespace boost {
  namespace sort {
    template<typename Data_type, typename Cast_type>
      Cast_type float_mem_cast(const Data_type &);
    template<typename RandomAccessIter>
      void float_sort(RandomAccessIter, RandomAccessIter);
    template<typename RandomAccessIter, typename Right_shift>
      void float_sort(RandomAccessIter, RandomAccessIter, Right_shift);
    template<typename RandomAccessIter, typename Right_shift,
             typename Compare>
      void float_sort(RandomAccessIter, RandomAccessIter, Right_shift,
                      Compare);
  }
}
```

Function template float_mem_cast

boost::sort::float_mem_cast — Casts a float to the specified integer type.

Synopsis

```
// In header: <boost/sort/spreadsort/float_sort.hpp>


template<typename Data_type, typename Cast_type>
  Cast_type float_mem_cast(const Data_type & data);
```

Description

Example:

```
struct rightshift {
  int operator()(const DATA_TYPE &x, const unsigned offset) const {
    return float_mem_cast<KEY_TYPE, CAST_TYPE>(x.key) >> offset;
  }
};
```

Template Parameters:	`Cast_type`	Integer type (same size) to which to cast.
	`Data_type`	Floating-point IEEE 754/IEC559 type.

Function template float_sort

boost::sort::float_sort — `float_sort` with casting to the appropriate size.

Synopsis

```
// In header: <boost/sort/spreadsort/float_sort.hpp>


template<typename RandomAccessIter>
  void float_sort(RandomAccessIter first, RandomAccessIter last);
```

Description

Some performance plots of runtime vs. n and log(range) are provided:
windows_float_sort
osx_float_sort

A simple example of sorting some floating-point is:

```
vector<float> vec;
vec.push_back(1.0);
vec.push_back(2.3);
vec.push_back(1.3);
spreadsort(vec.begin(), vec.end());
```

The sorted vector contains ascending values "1.0 1.3 2.3".

Parameters:	`first`	Iterator pointer to first element.
	`last`	Iterator pointing to one beyond the end of data.
Template Parameters:	`RandomAccessIter`	Random access iterator

Function template float_sort

boost::sort::float_sort — Floating-point sort algorithm using random access iterators with just right-shift functor.

Synopsis

```
// In header: <boost/sort/spreadsort/float_sort.hpp>


template<typename RandomAccessIter, typename Right_shift>
  void float_sort(RandomAccessIter first, RandomAccessIter last,
                  Right_shift rshift);
```

Description

Parameters:	`first`	Iterator pointer to first element.
	`last`	Iterator pointing to one beyond the end of data.
	`rshift`	Number of bits to right-shift (using functor).
Template Parameters:	`RandomAccessIter`	Random access iterator
	`Right_shift`	Functor for right-shift by parameter `shift` bits.

Function template float_sort

boost::sort::float_sort — Float sort algorithm using random access iterators with both right-shift and user-defined comparison operator.

Synopsis

```
// In header: <boost/sort/spreadsort/float_sort.hpp>


template<typename RandomAccessIter, typename Right_shift, typename Compare>
  void float_sort(RandomAccessIter first, RandomAccessIter last,
                  Right_shift rshift, Compare comp);
```

Description

Parameters:	`comp`	comparison functor.
	`first`	Iterator pointer to first element.
	`last`	Iterator pointing to one beyond the end of data.
	`rshift`	Number of bits to right-shift (using functor).
Template Parameters:	`RandomAccessIter`	Random access iterator
	`Right_shift`	functor for right-shift by parameter `shift` bits.

Header <boost/sort/spreadsort/integer_sort.hpp>

```
namespace boost {
  namespace sort {
    template<typename RandomAccessIter>
      void integer_sort(RandomAccessIter, RandomAccessIter);
    template<typename RandomAccessIter, typename Right_shift,
             typename Compare>
      void integer_sort(RandomAccessIter, RandomAccessIter, Right_shift,
                        Compare);
    template<typename RandomAccessIter, typename Right_shift>
      void integer_sort(RandomAccessIter, RandomAccessIter, Right_shift);
  }
}
```

Function template integer_sort

boost::sort::integer_sort — Integer sort algorithm using random access iterators. (All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

Synopsis

```
// In header: <boost/sort/spreadsort/integer_sort.hpp>


template<typename RandomAccessIter>
  void integer_sort(RandomAccessIter first, RandomAccessIter last);
```

Description

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).
Worst-case performance is $O(N * (lg(range)/s + s))$, so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is $O(N * ((32/11)$ slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_integer_sort

osx_integer_sort

> **Warning**
>
> Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

> **Warning**
>
> Invalid arguments cause undefined behaviour.

> **Note**
>
> `spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`first` Iterator pointer to first element. `last` Iterator pointing to one beyond the end of data.
Template Parameters:	`RandomAccessIter` Random access iterator
Requires:	`[first, last)` is a valid range.
Requires:	`RandomAccessIter` `value_type` is mutable.
Requires:	`RandomAccessIter` `value_type` is LessThanComparable
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.
Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Function template integer_sort

boost::sort::integer_sort — Integer sort algorithm using random access iterators with both right-shift and user-defined comparison operator. (All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

Synopsis

```
// In header: <boost/sort/spreadsort/integer_sort.hpp>


template<typename RandomAccessIter, typename Right_shift, typename Compare>
  void integer_sort(RandomAccessIter first, RandomAccessIter last,
                    Right_shift shift, Compare comp);
```

Description

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).
Worst-case performance is *O(N * (lg(range)/s + s))* , so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is *O(N * ((32/11)* slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_integer_sort
osx_integer_sort

> **Warning**
>
> Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

> **Warning**
>
> Invalid arguments cause undefined behaviour.

> **Note**
>
> `spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`comp` comparison functor. `first` Iterator pointer to first element. `last` Iterator pointing to one beyond the end of data. `shift` Number of bits to right-shift (using functor).
Template Parameters:	`RandomAccessIter` Random access iterator `Right_shift` functor for right-shift by parameter `shift` bits.
Requires:	`[first, last)` is a valid range.
Requires:	`RandomAccessIter` `value_type` is mutable.
Requires:	`RandomAccessIter` `value_type` is LessThanComparable
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.
Returns:	`void`.
Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Function template integer_sort

boost::sort::integer_sort — Integer sort algorithm using random access iterators with just right-shift functor. (All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

Synopsis

```
// In header: <boost/sort/spreadsort/integer_sort.hpp>


template<typename RandomAccessIter, typename Right_shift>
  void integer_sort(RandomAccessIter first, RandomAccessIter last,
                    Right_shift shift);
```

Description

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).

Performance: Worst-case performance is *O(N * (lg(range)/s + s))* , so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is *O(N * ((32/11)* slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_integer_sort
osx_integer_sort

Warning

Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

Warning

Invalid arguments cause undefined behaviour.

Note

`spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`first`	Iterator pointer to first element.
	`last`	Iterator pointing to one beyond the end of data.
	`shift`	Number of bits to right-shift (using functor).
Template Parameters:	`RandomAccessIter`	Random access iterator
	`Right_shift`	functor for right-shift by parameter `shift` bits.
Requires:	`[first, last)` is a valid range.	
Requires:	`RandomAccessIter` `value_type` is mutable.	
Requires:	`RandomAccessIter` `value_type` is LessThanComparable	
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.	
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.	

Throws: std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Header <boost/sort/spreadsort/spreadsort.hpp>

```
namespace boost {
  namespace sort {
    template<typename RandomAccessIter>
      boost::enable_if_c< std::numeric_limits< typename std::iterator_traits< RandomAc↵
cessIter >::value_type >::is_integer, void >::type
        spreadsort(RandomAccessIter, RandomAccessIter);
    template<typename RandomAccessIter>
      boost::enable_if_c< !std::numeric_limits< typename std::iterator_traits< RandomAc↵
cessIter >::value_type >::is_integer &&std::numeric_limits< typename std::iterat↵
or_traits< RandomAccessIter >::value_type >::is_iec559, void >::type
        spreadsort(RandomAccessIter, RandomAccessIter);
    template<typename RandomAccessIter>
        boost::enable_if_c< is_same< typename std::iterator_traits< RandomAcces↵
sIter >::value_type, typename std::string >::value||is_same< typename std::iterat↵
or_traits< RandomAccessIter >::value_type, typename std::wstring >::value, void >::type
        spreadsort(RandomAccessIter, RandomAccessIter);
  }
}
```

Function template spreadsort

boost::sort::spreadsort — Generic `spreadsort` variant detecting integer-type elements so call to `integer_sort`.

Synopsis

```
// In header: <boost/sort/spreadsort/spreadsort.hpp>


template<typename RandomAccessIter>
  boost::enable_if_c< std::numeric_limits< typename std::iterator_traits< RandomAcces↵
sIter >::value_type >::is_integer, void >::type
  spreadsort(RandomAccessIter first, RandomAccessIter last);
```

Description

If the data type provided is an integer, `integer_sort` is used.

> **Note**
>
> Sorting other data types requires picking between `integer_sort`, `float_sort` and `string_sort` directly, as `spreadsort` won't accept types that don't have the appropriate `type_traits`.

Parameters:
- `first` Iterator pointer to first element.
- `last` Iterator pointing to one beyond the end of data.

Template Parameters: `RandomAccessIter` Random access iterator

Requires: `[first, last)` is a valid range.

Requires: `RandomAccessIter` `value_type` is mutable.

Requires: `RandomAccessIter` `value_type` is LessThanComparable

Requires: `RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.

Postconditions:	The elements in the range [`first`, `last`) are sorted in ascending order.

Function template spreadsort

boost::sort::spreadsort — Generic `spreadsort` variant detecting float element type so call to `float_sort`.

Synopsis

```
// In header: <boost/sort/spreadsort/spreadsort.hpp>


template<typename RandomAccessIter> 
  boost::enable_if_c< !std::numeric_limits< typename std::iterator_traits< RandomAcces↵
sIter >::value_type >::is_integer &&std::numeric_limits< typename std::iterat↵
or_traits< RandomAccessIter >::value_type >::is_iec559, void >::type 
  spreadsort(RandomAccessIter first, RandomAccessIter last);
```

Description

If the data type provided is a float or castable-float, `float_sort` is used.

> **Note**
>
> Sorting other data types requires picking between `integer_sort`, `float_sort` and `string_sort` directly, as `spreadsort` won't accept types that don't have the appropriate `type_traits`.

Parameters:	`first`	Iterator pointer to first element.
	`last`	Iterator pointing to one beyond the end of data.
Template Parameters:	`RandomAccessIter`	Random access iterator
Requires:	[`first`, `last`) is a valid range.	
Requires:	`RandomAccessIter` `value_type` is mutable.	
Requires:	`RandomAccessIter` `value_type` is LessThanComparable	
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.	
Postconditions:	The elements in the range [`first`, `last`) are sorted in ascending order.	

Function template spreadsort

boost::sort::spreadsort — Generic `spreadsort` variant detecting string element type so call to `string_sort` for `std::strings` and `std::wstrings`.

Synopsis

```
// In header: <boost/sort/spreadsort/spreadsort.hpp>


template<typename RandomAccessIter> 
  boost::enable_if_c< is_same< typename std::iterator_traits< RandomAcces↵
sIter >::value_type, typename std::string >::value||is_same< typename std::iterat↵
or_traits< RandomAccessIter >::value_type, typename std::wstring >::value, void >::type 
  spreadsort(RandomAccessIter first, RandomAccessIter last);
```

Description

If the data type provided is a string or wstring, `string_sort` is used.

> **Note**
>
> Sorting other data types requires picking between `integer_sort`, `float_sort` and `string_sort` directly, as `spreadsort` won't accept types that don't have the appropriate `type_traits`.

Parameters:	`first` Iterator pointer to first element. `last` Iterator pointing to one beyond the end of data.
Template Parameters:	`RandomAccessIter` Random access iterator
Requires:	`[first, last)` is a valid range.
Requires:	`RandomAccessIter` `value_type` is mutable.
Requires:	`RandomAccessIter` `value_type` is LessThanComparable
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.

Header <boost/sort/spreadsort/string_sort.hpp>

```
namespace boost {
  namespace sort {
    template<typename RandomAccessIter, typename Unsigned_char_type>
      void string_sort(RandomAccessIter, RandomAccessIter, Unsigned_char_type);
    template<typename RandomAccessIter>
      void string_sort(RandomAccessIter, RandomAccessIter);
    template<typename RandomAccessIter, typename Compare,
             typename Unsigned_char_type>
      void reverse_string_sort(RandomAccessIter, RandomAccessIter, Compare,
                               Unsigned_char_type);
    template<typename RandomAccessIter, typename Compare>
      void reverse_string_sort(RandomAccessIter, RandomAccessIter, Compare);
    template<typename RandomAccessIter, typename Get_char,
             typename Get_length>
      void string_sort(RandomAccessIter, RandomAccessIter, Get_char,
                       Get_length);
    template<typename RandomAccessIter, typename Get_char,
             typename Get_length, typename Compare>
      void string_sort(RandomAccessIter, RandomAccessIter, Get_char,
                       Get_length, Compare);
    template<typename RandomAccessIter, typename Get_char,
             typename Get_length, typename Compare>
      void reverse_string_sort(RandomAccessIter, RandomAccessIter, Get_char,
                               Get_length, Compare);
  }
}
```

Function template string_sort

boost::sort::string_sort — String sort algorithm using random access iterators, allowing character-type overloads. (All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

Synopsis

```
// In header: <boost/sort/spreadsort/string_sort.hpp>


template<typename RandomAccessIter, typename Unsigned_char_type>
  void string_sort(RandomAccessIter first, RandomAccessIter last,
                   Unsigned_char_type unused);
```

Description

`string_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).

Worst-case performance is $O(N * (lg(range)/s + s))$, so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is $O(N * ((32/11)$ slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_string_sort
osx_string_sort

Warning

Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

Warning

Invalid arguments cause undefined behaviour.

Note

`spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of $O(N*log(N))$ comparisons and $O(N*log(K/S + S))$ operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`first`	Iterator pointer to first element.
	`last`	Iterator pointing to one beyond the end of data.
	`unused`	Unused ???
Template Parameters:	`RandomAccessIter`	Random access iterator
	`Unsigned_char_type`	Unsigned character type used for string.
Requires:	`[first, last)` is a valid range.	
Requires:	`RandomAccessIter value_type` is mutable.	
Requires:	`RandomAccessIter value_type` is LessThanComparable	
Requires:	`RandomAccessIter value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.	
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.	

Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Function template string_sort

boost::sort::string_sort — String sort algorithm using random access iterators, wraps using default of unsigned char. (All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

Synopsis

```
// In header: <boost/sort/spreadsort/string_sort.hpp>


template<typename RandomAccessIter>
  void string_sort(RandomAccessIter first, RandomAccessIter last);
```

Description

`string_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).
Worst-case performance is *O(N * (lg(range)/s + s))* , so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is *O(N * ((32/11)* slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_string_sort
osx_string_sort

Warning

Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

Warning

Invalid arguments cause undefined behaviour.

Note

`spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`first`	Iterator pointer to first element.
	`last`	Iterator pointing to one beyond the end of data.
Template Parameters:	`RandomAccessIter`	Random access iterator
Requires:	`[first, last)` is a valid range.	

Requires:	`RandomAccessIter` `value_type` is mutable.
Requires:	`RandomAccessIter` `value_type` is LessThanComparable
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.
Postconditions:	The elements in the range [`first`, `last`) are sorted in ascending order.
Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Function template reverse_string_sort

boost::sort::reverse_string_sort — String sort algorithm using random access iterators, allowing character-type overloads.

Synopsis

```
// In header: <boost/sort/spreadsort/string_sort.hpp>


template<typename RandomAccessIter, typename Compare,
         typename Unsigned_char_type>
  void reverse_string_sort(RandomAccessIter first, RandomAccessIter last,
                           Compare comp, Unsigned_char_type unused);
```

Description

(All variants fall back to `std::sort` if the data size is too small, < detail::min_sort_size).

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).
Worst-case performance is *O(N * (lg(range)/s + s))* , so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is *O(N * ((32/11)* slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_integer_sort
osx_integer_sort

Warning

Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

Warning

Invalid arguments cause undefined behaviour.

Note

`spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`comp` comparison functor. `first` Iterator pointer to first element. `last` Iterator pointing to one beyond the end of data. `unused` Unused ???
Template Parameters:	`RandomAccessIter` Random access iterator `Unsigned_char_type` Unsigned character type used for string.
Requires:	`[first, last)` is a valid range.
Requires:	`RandomAccessIter` `value_type` is mutable.
Requires:	`RandomAccessIter` `value_type` is LessThanComparable
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.
Returns:	`void`.
Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Function template reverse_string_sort

boost::sort::reverse_string_sort — String sort algorithm using random access iterators, wraps using default of `unsigned` char.

Synopsis

```
// In header: <boost/sort/spreadsort/string_sort.hpp>


template<typename RandomAccessIter, typename Compare>
  void reverse_string_sort(RandomAccessIter first, RandomAccessIter last,
                           Compare comp);
```

Description

(All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).
Worst-case performance is *O(N * (lg(range)/s + s))* , so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is *O(N * ((32/11)* slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_integer_sort
osx_integer_sort

Warning

Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

> **Warning**
>
> Invalid arguments cause undefined behaviour.

> **Note**
>
> `spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`comp` Comparison functor. `first` Iterator pointer to first element. `last` Iterator pointing to one beyond the end of data.
Template Parameters:	`RandomAccessIter` Random access iterator
Requires:	`[first, last)` is a valid range.
Requires:	`RandomAccessIter` `value_type` is mutable.
Requires:	`RandomAccessIter` `value_type` is LessThanComparable
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.
Returns:	`void`.
Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Function template string_sort

boost::sort::string_sort — String sort algorithm using random access iterators, wraps using default of `unsigned` char.

Synopsis

```
// In header: <boost/sort/spreadsort/string_sort.hpp>


template<typename RandomAccessIter, typename Get_char, typename Get_length>
  void string_sort(RandomAccessIter first, RandomAccessIter last,
                   Get_char getchar, Get_length length);
```

Description

(All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).
Worst-case performance is *O(N * (lg(range)/s + s))* , so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is *O(N * ((32/11)* slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:

windows_integer_sort
osx_integer_sort

> **Warning**
>
> Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

> **Warning**
>
> Invalid arguments cause undefined behaviour.

> **Note**
>
> `spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`first`	Iterator pointer to first element.
	`getchar`	Number corresponding to the character offset from bracket functor equivalent to `operator`[].
	`last`	Iterator pointing to one beyond the end of data.
	`length`	Functor to get the length of the string in characters.
Template Parameters:	`Get_char`	Bracket functor equivalent to `operator`[], taking a number corresponding to the character offset..
	`Get_length`	Functor to get the length of the string in characters. TODO Check this and below and other places!!!
	`RandomAccessIter`	Random access iterator
Requires:	`[first, last)` is a valid range.	
Requires:	`RandomAccessIter` `value_type` is mutable.	
Requires:	`RandomAccessIter` `value_type` is LessThanComparable	
Requires:	`RandomAccessIter` `value_type` supports the `operator>>`, which returns an integer-type right-shifted a specified number of bits.	
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.	
Returns:	`void`.	
Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.	

Function template string_sort

boost::sort::string_sort — String sort algorithm using random access iterators, wraps using default of `unsigned` char.

Synopsis

```
// In header: <boost/sort/spreadsort/string_sort.hpp>


template<typename RandomAccessIter, typename Get_char, typename Get_length,
         typename Compare>
  void string_sort(RandomAccessIter first, RandomAccessIter last,
                   Get_char getchar, Get_length length, Compare comp);
```

Description

(All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).
Worst-case performance is *O(N * (lg(range)/s + s))* , so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is *O(N * ((32/11)* slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_integer_sort
osx_integer_sort

Warning

Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

Warning

Invalid arguments cause undefined behaviour.

Note

`spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`comp`	comparison functor.
	`first`	Iterator pointer to first element.
	`getchar`	???
	`last`	Iterator pointing to one beyond the end of data.
	`length`	???
Template Parameters:	`Get_char`	???.
	`Get_length`	??? TODO
	`RandomAccessIter`	Random access iterator
Requires:	`[first, last)` is a valid range.	

Requires:	`RandomAccessIter` `value_type` is mutable.
Requires:	`RandomAccessIter` `value_type` is LessThanComparable
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.
Returns:	`void`.
Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Function template reverse_string_sort

boost::sort::reverse_string_sort — Reverse String sort algorithm using random access iterators.

Synopsis

```
// In header: <boost/sort/spreadsort/string_sort.hpp>


template<typename RandomAccessIter, typename Get_char, typename Get_length,
         typename Compare>
  void reverse_string_sort(RandomAccessIter first, RandomAccessIter last,
                           Get_char getchar, Get_length length, Compare comp);
```

Description

(All variants fall back to `std::sort` if the data size is too small, < `detail::min_sort_size`).

`integer_sort` is a fast templated in-place hybrid radix/comparison algorithm, which in testing tends to be roughly 50% to 2X faster than `std::sort` for large tests (>=100kB).
Worst-case performance is *O(N * (lg(range)/s + s))* , so `integer_sort` is asymptotically faster than pure comparison-based algorithms. `s` is `max_splits`, which defaults to 11, so its worst-case with default settings for 32-bit integers is *O(N * ((32/11)* slow radix-based iterations fast comparison-based iterations).
Some performance plots of runtime vs. n and log(range) are provided:
windows_integer_sort
osx_integer_sort

Warning

Throwing an exception may cause data loss. This will also throw if a small vector resize throws, in which case there will be no data loss.

Warning

Invalid arguments cause undefined behaviour.

Note

`spreadsort` function provides a wrapper that calls the fastest sorting algorithm available for a data type, enabling faster generic-programming.

The lesser of *O(N*log(N))* comparisons and *O(N*log(K/S + S))* operations worst-case, where:

* N is `last` - `first`,

* K is the log of the range in bits (32 for 32-bit integers using their full range),

* S is a constant called max_splits, defaulting to 11 (except for strings where it is the log of the character size).

Parameters:	`comp`	comparison functor.
	`first`	Iterator pointer to first element.
	`getchar`	???
	`last`	Iterator pointing to one beyond the end of data.
	`length`	???
Template Parameters:	`Get_char`	???.
	`Get_length`	??? TODO
	`RandomAccessIter`	Random access iterator

Requires:	`[first, last)` is a valid range.
Requires:	`RandomAccessIter` `value_type` is mutable.
Requires:	`RandomAccessIter` `value_type` is LessThanComparable
Postconditions:	The elements in the range `[first, last)` are sorted in ascending order.
Returns:	`void`.
Throws:	std::exception Propagates exceptions if any of the element comparisons, the element swaps (or moves), the right shift, subtraction of right-shifted elements, functors, or any operations on iterators throw.

Function Index

Index