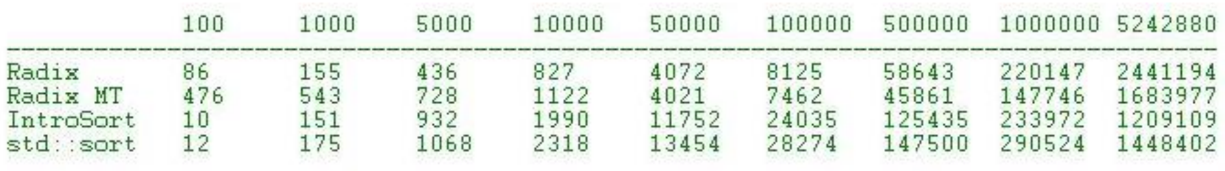
Radix Redux

Pierre Terdiman, 2018, v1.0

Again ?

Back in 2000 I published a short article about radix sort on my website (1). The main goal of the article was to show how to handle floating-point values with a radix sort, which is something that may not have been entirely obvious at the time.

Over the years, I have received multiple emails pointing out that my radix sort was sometimes slower than traditional sorting algorithms. This is correct and not a surprise. In my initial article I was already mentioning that radix sort is not a good choice for sorting a small number of values. Then in 2008 I wrote a blog post (2) which had a small picture, reproduced here, summarizing the performance of various sorts for various numbers of elements (3):



```
             100     1000    5000    10000   50000   100000  500000  1000000 5242880
------------------------------------------------------------------------------------
Radix        86      155     436     827     4072    8125    58643   220147  2441194
Radix MT     476     543     728     1122    4021    7462    45861   147746  1683977
IntroSort    10      151     932     1990    11752   24035   125435  233972  1209109
std::sort    12      175     1068    2318    13454   28274   147500  290524  1448402
```

In these tests, “Radix” was an implementation similar to the one I published in 2000. As you could see from these results, for a small number of values (~100) or for a very large number of values (>1 million), alternative algorithms were indeed faster. (I ignore the multi-threaded Radix experiment here, just to compare apples to apples). This is old news really.

Now, in 2018, about 18 years after I published that article, somebody once again contacted me (on Facebook, of all places) to tell me that some other implementation online was faster than mine when sorting 10 million values.

Yes. I know. That is quite enough now.

I was explaining for the Nth time what the issue was with the radix code when I figured that maybe it would be a good idea to put the information once and for all on my blog.

So, there.

The issue

A radix sort uses more memory than e.g. a regular quick-sort implementation, so unsurprisingly the issue is cache misses. All things being otherwise equal, using more memory naturally gives birth to more cache misses when you access that extra memory. But what matters even more than the amount of memory used is *how* you access that memory: sequentially or randomly. In that respect my old implementation was pretty bad.

Grab the original code and just look at it:

```
while(Indices!=IndicesEnd)
{
        udword id = *Indices++;
        mIndices2[mOffset[InputBytes[id<<2]]++] = id;
}
```

There are multiple reads and writes here:

- We parse the previous array of sorted indices (a.k.a. ranks):

  ```
  udword id = *Indices++;
  ```

  Status: green. This is a simple sequential read, so no problem here.

- Then using that previous rank, we read the input bytes:

  ```
  InputBytes[id<<2]
  ```

  Status: red. This is a completely random read access from a potentially large array. Costly.

- Then using that byte, we update (read/write) the offsets:

  ```
  mOffset[…]++
  ```

  Status: orange. This is a random read/modify/write operation in a small array. This is random access but it will most likely “always” be in the cache, so probably not that bad.

- Then using the offset, we write the next ranks:

  ```
  mIndices2[…] = id;
  ```

  Status: red. This is a random write access to a potentially large array. Costly. It might be a little bit less costly than the previous red read operation, because the number of target addresses is limited, and we only write to them sequentially. But still, pretty bad.

So the sorting loop does not do a lot of work, but the few operations it does can potentially be costly.

Indices vs values

The code was written this way because it is a “rank sorter”, which means it returns an array of sorted *indices*, or *ranks*:

```
	//! Access to results. mIndices is a list of indices in sorted order, i.e. in the
order you may further process your data
	inline_		udword*		GetIndices()	const	{ return mIndices;	}
```

These ranks are indexing the initial array of input *values*, which remains untouched. That’s what the const in the function’s signature tells you:

```
	RadixSort&	Sort(const float* input, udword nb);
```

Traditional algorithms like *std::sort()* work differently. By default they are “value sorters”, which means they do modify the array of passed values (they sort it):

```
	int* values;  // Array of N values to sort
	std::sort(values, values+N);// Values have been reshuffled/sorted after the call
```

They don’t have a built-in notion of ranks: if you need ranks, you need to add and manage them yourself.

```
	struct Key
	{
		int		mValue;
		int		mRank;

		bool	operator==(const Key& p)	const	{ return mValue == p.mValue;}
		bool	operator<=(const Key& p)	const	{ return mValue <= p.mValue }
		bool	operator>=(const Key& p)	const	{ return mValue >= p.mValue }
		bool	operator<(const Key& p)		const	{ return mValue < p.mValue; }
		bool	operator>(const Key& p)		const	{ return mValue > p.mValue; }
	};

	Key* values;  // Array of N keys to sort

	std::sort(values, values+N); ); // Keys have been reshuffled/sorted after the call
```

That last call will reshuffle the keys, i.e. both the values and the ranks. Technically the algorithm does not need the ranks, but they are useful for users to make sense of the results. For example if you are sorting objects by distance, you are not really interested in the actual list of sorted distance values: what you want is a list of sorted objects, an *order* in which you will process them. My radix sort implementation recognized that fact and avoided modifying the values at all.

But this design choice had a direct impact on performance and memory usage. In a nutshell, dropping the values entirely limits the amount of extra memory used by the sort function, but increases the number of cache misses and makes performance worse.

Let’s revisit this design choice today and see what we come up with.

An alternative implementation

A straightforward way to improve the code is to output *both* the rank and value from the sorting loop – like what the *std::sort()* function does when given an array of "keys". This helps because the two first memory accesses that we looked at are actually just retrieving the values in their sorted order so far.

This means that we can replace this:

```
udword id = *Indices++;     // green
InputBytes[id<<2]           // red
```

…with just a sequential read access (green) of values output in the previous pass.

Now there is a trap here: as we just said we need to output an extra value now, like we output the ranks in the initial code. Which was done this way:

```
mIndices2[…] = id;   // red
```

This is *also* a red / costly write. The trap (and a common mistake if I look a radix sort implementations on the internet) is to output the value independently of the rank, to a *separate* buffer. If you use a separate buffer, you're getting two cache misses instead of one and replacing a costly read access with a costly write access: not great. The right thing to do here is to copy what we had for the *std::sort()* call, and mix the rank and value in the same "Key" or "combo" structure. That way there is only one target address to write to, as before, and while we do write twice as much data as in the previous code we at least avoid getting two cache misses.

The next trick then, is to recognize that the values are only needed to make the read accesses in the next pass faster. Thus, we do not need to output them in the last pass. We can output the ranks alone in the last pass, which allows us to present the results to users in the same way as for the initial implementation: just an array of sorted indices.

There are a lot of other minor implementation details that could be discussed (like using templates to hardcode the pass index and make the loop a bit tighter). Please refer to the companion source code if you are interested.

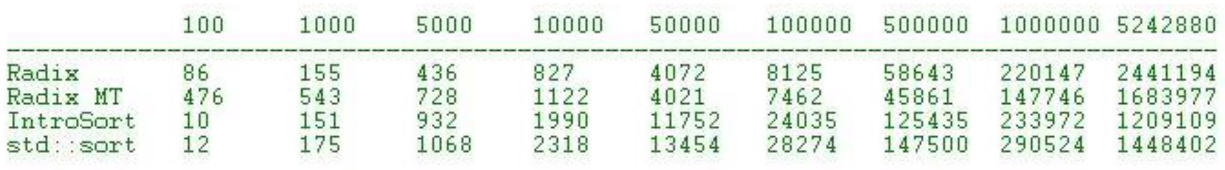
The only clear drawback for this alternative implementation is that it needs more memory internally, to store the sorted values (that we did not need to store before).

Note that this new implementation is a proof-of-concept prototype that does not re-implement all the features of the original code. In particular it only sorts integers (not floats) and it does not support what we called "temporal coherence" in the initial article.

Results

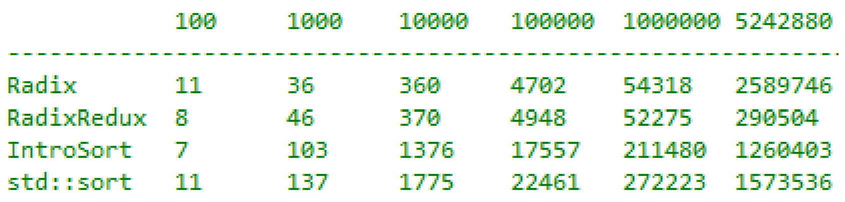
I used my old test project (2), recompiled with a new compiler (*Visual Studio 2015*) on a new machine (i7-based). I removed the multi-threaded radix experiment and added a new, more cache-friendly radix sort implementation ("RadixRedux") based on this article.

In 2008 I got:



```
             100      1000     5000     10000    50000    100000   500000   1000000  5242880
-----------------------------------------------------------------------------------------------
Radix        86       155      436      827      4072     8125     58643    220147   2441194
Radix MT     476      543      728      1122     4021     7462     45861    147746   1683977
IntroSort    10       151      932      1990     11752    24035    125435   233972   1209109
std::sort    12       175      1068     2318     13454    28274    147500   290524   1448402
```

In 2018 I got:



```
            100     1000    10000   100000  1000000 5242880
-----------------------------------------------------------

Radix       11      36      360     4702    54318   2589746
RadixRedux  8       46      370     4948    52275   290504
IntroSort   7       103     1376    17557   211480  1260403
std::sort   11      137     1775    22461   272223  1573536
```

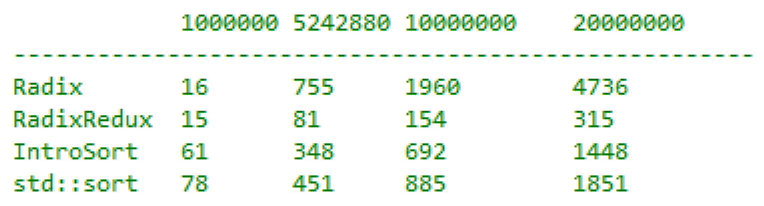
The top numbers are the number of values to sort. The other numbers next to the sorts' names are the timings in K-cycles. We see that:

- For all implementations the same code became faster in 2018 compared to 2008. This is expected when using a new compiler on a new machine but it's good to see it does indeed happen.

- The radix sort code seems to benefit more from this new environment. The relative performance improvements due to just recompiling the code seem higher for the radix sort than for the alternative sorts. It might be because the new test is sorting integers while the old test was sorting floats (which goes to a special codepath in the radix implementation). Or it might be that the SSE2 compile flag gives better results on the radix code. Or something. I did not investigate.

- Up to a million values, there isn't much difference between the two radix sort implementations. On the other hand what is not shown here is that the new implementation uses more memory, to store temporary sorted values. Typically games and other interactive applications do not need to sort a million values at runtime, so for them the regular implementation is probably better. This validates the design decision made 18 years ago in the initial implementation.

Beyond a million values the times as reported by *rdtsc()* became a bit dubious. I think I started to overflow the 32bit part of the TSC counter. Just to be on the safe side, I switched to *timeGetTime()* and repeated the benchmarks for large numbers of values. The following numbers are now in ms:



```
            1000000 5242880 10000000    20000000
-----------------------------------------------------
Radix       16      755     1960        4736
RadixRedux  15      81      154         315
IntroSort   61      348     692         1448
std::sort   78      451     885         1851
```

We see that beyond a million values to sort, the new implementation is significantly faster than the old one – sometimes more than an order of magnitude faster. This is because it is more cache-friendly, which helps when we run out of cache.

We also see that this new version makes the radix sort faster than alternative implementations again.

Conclusion

This article explored the limits of my “radix sort revisited” implementation. It showed that this old version does indeed becomes slower than traditional sorting algorithms for a large number of values.

We showed how to overcome these limits and discussed implementation details to make the code more cache-friendly and sometimes significantly faster. At the same time we saw that the new version does not perform better than the old one for “reasonable” numbers of values. But it does use more memory, so the initial design decision for the code was perhaps the right one.

See you in 18 more years ☺

References

1) http://www.codercorner.com/RadixSortRevisited.htm
2) http://www.codercorner.com/blog/?p=90
3) http://www.codercorner.com/Pictures/Sorts.jpg