Last Updated: October 11, 2021

Write-up 1

Execution time and instruction references for `DEBUG=1` is close to double `DEBUG=0`. There are approximately three times more data references.

The number of instruction references can't really be substituted for execution time when measuring performance. Different instructions take different amounts of time (e.g. addition is faster than multiplication).

Write-up 2

You can expect to see a very small performance improvement ($\sim$ 3%) from inlining `merge_i` and `copy_i`. You can try to inline the recursive function `sort_i`, but if you look at the annotated assembly you'll see that clang does not inline it.

Write-up 3

With link-time-optimization enabled, you can once again see a slight performance improvement of about $\sim$ 3% (for both `sort_a` and `sort_i`, since each will gain the benefit of LTO).

If you compare the perf reports before and after, you'll notice that the code for `mem_alloc` and `mem_free` are pasted in the body of the report after LTO, and there's no longer the telltale `callq` instruction indicating a function call. In addition, you'll notice that the `mem_alloc` and `mem_free` entries are removed from the main list of symbols.

Write-up 4

The algorithm no longer has the extra overhead of managing these counters, since it can perform arithmetic directly on the pointer values.

This gives a slight performance improvement ($\sim$ 3%).

Write-up 5

You can coarsen the recursion by checking whether the number of elements is under a certain threshold, and then calling a different sort routine if it is. If you used the staff insertion sort, a good threshold value is anywhere between 30 and 70. Insertion sort is faster for small numbers because there's less overhead the merge sort, but as the number of elements increases, insertion sort becomes asymptotically worse.

You should see a big performance benefit from coarsening (on the order of $\sim 50\%$).

Write-up 6

You can eliminate one of your temporary arrays by using the second half of the original array as "scratch space." As you select the smallest items from the two lists and write them into the original array, you won't overwrite any data you need to keep.

A compiler would not be able to make this optimization for you, as the fact that the optimization can be done at all relies on an algorithmic invariant.

In particular, suppose we're in the middle of a merge of two arrays, A and B, and the output of the merge goes into C. The current index for A is i and the current index for B is j, so the current index for C is $i + j$. If we've used the right-half of C for B, then since $i \leq |A|$ we have $i + j \leq |A| + j$, i.e. the pointer into C will never surpass the pointer into B. The merge routine may start overwriting B, but that part of B will not be used by the remainder of the merge. A compiler would not be able to notice this invariant, and would only see that arrays B and C are in use simultaneously at best.

Using this invariant moreover requires another change to the code; sentinels are used to mark the ends of the arrays A and B, but a sentinel cannot be used for B if it is stored at the right-end of C, since C would be one element too short. To fix this, a check needs to be added to ensure that the right pointer hasn't fallen off the end of B before dereferencing the right pointer. The compiler would likely be unable to notice this as well.

There should be a modest performance benefit after eliminating half your scratch space ($\sim 20\%$), since allocating memory can be expensive.

Write-up 7

You should notice a significant performance benefit ($\sim$ 40%) to reusing scratch space, since allocating memory can be expensive.