Review of "Sort Race" – SPE-21-007.R1

Summary

The paper is greatly improved but it still has a number of issues.

Issues

- I want to see a discussion of the different kinds of arrays that might be sorted. There are really three kinds to consider (where I use C to give examples of what I mean):
  1. arrays of simple values, such as `int A[100];`
  2. arrays of structured values, such as `struct{char *name; int age;} B[100];`
  3. arrays of references, such as `struct{char *name; int age;} *C[100];`

  The Java language provides only the first and third of these possibilities, while C# and C++ provide all three.

  With array A above, the compiler can generate efficient code for a sort function where the machine's comparison instruction is used to compare two elements directly, where exchanging two elements can be performed using a register for the swap, and where several adjacent elements of the array will be in the same cache line and so reduce the number of cache misses while sorting.

  With array B, the compiler may not be able to generate efficient code for comparisons, swapping two elements will be expensive especially if byte-by-byte memory moves are used, and adjacent elements may not be sharing the same cache line. A common strategy is to construct a temporary array of pointers to these elements (so it is of the third array type shown above), sort the temporary array, and then use it to move the elements of the original array into sorted order. This strategy minimizes the number of memory moves needed.

  With array C, there is an extra memory indirection needed to access an array element (making comparisons more expensive) and the elements themselves are likely to be scattered all over heap memory, leading to a high cache miss rate.

  It should be pointed out that these considerations are very likely to affect which sorting method will be fastest.
- On page 3, we are told that one possibility for an array element is lists of integers (of lengths 64, 256 and 1024 bytes). Please be more precise about what that means. Do the lists themselves appear as array elements or do you mean that the array contains pointers to the lists? In C, this would be the difference between array A1 and array A2 below:

```
typedef char ELEMTYPE[64];  // used for lists of 64 bytes
ELEMTYPE A1[100000];
(*ELEMTYPE) A2[100000];
```

  (Of course, A1 is really a two-dimensional matrix but where we consider each row of the matrix to be an element of the array.)
- Table 1 provides some running times but we are not told what the datatype of the array elements is. Are these arrays of simple integers? Please tell the reader, and please tell the reader if the best algorithm in each row depends on the element type. (And if the best algorithm is found to be, say, timsort for some element type, a further discussion is needed.)
- Page 2 uses the term "the latest *qsort*". I don't know where in the paper we are told which is the latest qsort. In any case, next year some even newer version of qsort might come into use and the reader will not know what you are referring to. What the paper needs is a table which gives a very

brief description and a reference (plus, when possible, a reference for the implementation) for every sort program which is mentioned anywhere in the paper. And then the narrative in the paper refers to the appropriate row in that table whenever a particular version of a sort program is mentioned. For example, the paper might provide a table like this:

Table 1: Sort Programs

#	Name	Description	Reference
1	B&M	Bentley & McIlroy *qsort*	[8]
2	qsort	Sort function in Gnu C library	[17]
...	...	...	

where [17] is a new reference which refers to https://code.woboq.org/userspace/glibc/stdlib/qsort.c.html . The text of the paper would refer to a particular version of a sort program like this *qsort* (Table 1, row 2). The sort functions created by the authors should not be included in this table.

- Page 4 makes the statement that "Mergesort is the best method for ...". This was only true for your experiments and other people using different C compilers on other computing platforms might observe something different. You have to be careful to word such statements in a similar manner to this: "In our experiments, we found mergesort to be the best method for ..."

Other Issues

- The paper provides a URL where the C source code can be downloaded. This URL appears to be referencing a student account at the University of Iowa. While it is excellent that readers are given access to the source code, a more permanent on-line repository for the source code must be used. For example, a GitHub repository would be a good choice for storing the code.
- As a matter of style, no URLs should appear in the body of the paper. The proper place would be in the references section.
- On page 2, it is stated that '`-O3`' was used as optimization level in gcc. I suggest that the authors re-run the speed tests with the '`-O2`' optimization level. In my own experience, this often produces faster code.

Major LaTeX Issue

- The authors have misused LaTeX commands when emphasized text is desired. On the left is one of the worst examples that occurs in the paper, and on the right is the way that it *should* appear:

Notice how the spacing of the characters is quite different, particularly around the letter '*f*.[1]
The horrible formatting on the left is caused by the misuse of math mode to generate emphasized text. The left-side example was created with the LaTeX input

1. This weird spacing occurs because math mode interprets 's', 'h', 'u', ... as being the names of variables which are to be multiplied together.

```
$k$-$shuffled$ $teeth$
```

and the right-side example was created with correct LaTeX usage

```
\textit{k-shuffled teeth}
```

[There are more ways in LaTeX to produce the desired result, but `\textit` is the usual command.]

The authors must ensure that all emphasized text has been created with the proper LaTeX commands before submitting a revised version.

Typographical Errors, etc.

- Abstract: "efficient sorting method" ⇒ "efficient sorting methods"
- Page 3: "miss fire" ⇒ "misfire"
- Page 5: "small impact to the overall performance" ⇒ "small impact on the overall performance"
- Psge 7: "the BSD sort function also use" ⇒ "the BSD sort function also uses"
- Page 7: "combing (e) and (f)" ⇒ "combining (e) and (f)"
- Page 8: the last column of Table 4 is missing a heading
- Page 8: "conducted an extensive experimentation on" ⇒ "conducted extensive experiments on"
- Page 10: Fix the equation "f(n) = 1.09nlog(n)λ0.76n"
  Also, it appears to me that there is another misuse of LaTeX here. Please use math mode for the equation and use `\log` to generate the log function name. (Be sure to use `\log` for all occurrences of the log function in the paper.)

The Source Code Files

I downloaded the very much improved source code from `https://drive.google.com/drive/folders/1kLYZnJpF6GNvnXM04QUxm0QXWmdWkPBs`.

Before the code can be shared with readers of the journal, some improvements and corrections should be made.

- There should be no warning messages when the files are compiled. I got many warning messages like this one when using gcc:

```
sort.c:1637:23: warning: format '%d' expects argument of type
 'int', but argument 3 has type 'long unsigned int' [-Wformat=]
  fprintf(stderr, "struct var * size (%d) != long size\n",
   sizeof(item64 *)
```

  You can make these messages go away by either using the format code (`%ld`) that `fprintf` is expecting, or you can cast the value to be printed to type `int`.
- All the shell scripts assume a particular path for accessing the `mysort` program, this path being `~/sort/mysort` . That is poor practice. At the very least, the path to `mysort` should be obtained from a shell variable.

  My preference would be for the main function (in `sort.c`) to accept one extra command line argument which makes all the shell scripts unnecessary. For example, invoking `mysort` with a command line like this:

```
mysort -reps=10 10000000 5 2
```

  could have the same effect as

```
repeat10 100000 5 2
```

  And

```
mysort -reps=10 100000 5
```

could have the same effect as

```
runtype 100000
```

- The `README` file implies that running the command `make` creates the optimized version of `mysort`. Unfortunately it does not, it creates the debug version because `debug` appears as a target before `sort` in the Makefile. While the command

  ```
  make sort
  ```

  could be used to force creation of the optimized version, it would be nice if the optimized version were the default.
- To build the production version of `mysort`, the `-O3` flag needs to be supplied when compiling each `.c` file and when linking the `.o` files in the last step. To build the debug version, the `-g` flag needs to be supplied in all these steps instead. The comments at the top of the Makefile should make it clear how to create either version of `mysort`.
- A good Makefile would contain an additional target named `clean` for removing superfluous files once the program has been built. I.e.,

  ```
  clean:          # remove superfluous files
          rm -f *.o core
  ```

- The README file should contain additional instructions which explain how to add a new sort program to the collection of programs evaluated by `mysort`.