Language Wars Assignment 3

By: Mike Lang
April 8th 2016

Overview

The purpose of this assignment is to code an algorithm in at least 4 languages and then compare and contrast your experience in each language. The 5 languages that i've chosen are C, D, Lua, Julia and Python. This should give me an interesting mix of languages because 2 of them are compiled languages, and 3 of them are interpreted languages. Experiments performed will include timing the execution of each language, as well as other comparisons.

Algorithm

The algorithm that i've chosen to work with is quicksort. I chose quicksort because it is known to be one of the fastest sorting algorithms on average. It was developed by Tony Hoare in 1959. Quicksort is an in-place sorting algorithm, which means that it works within the memory of the original array, swapping values around without the need of allocating any more memory. Quicksort has an average case complexity of O(n log n), and a worst case performance of $O(n^2)$. Although merge sort has a worst case complexity of O(n log n) compared to quicksort's $O(n^2)$, quicksort is known to be 2-3 times faster on average.

Quicksort is a divide-and-conquer algorithm, this means that it recursively breaks the problem down into smaller pieces until the problem becomes trivial to solve.

Quicksort does this by:

1. Choose an element to be the pivot point.
2. Reorder the array so that each element less than the partition is moved left of the pivot, and each element greater than the pivot is moved right of the pivot. The pivot point is now in its final position.
3. Recursively apply the above steps to the set of values to the left of the pivot and to the right of the pivot.

Issues:

The choice of pivot greatly affects the performance of the algorithm. For example, choosing the leftmost value to be the pivot point causes $O(n^2)$ performance on already sorted lists. Choosing a random point for the pivot or choosing the middle value is often suggested.

Lots of repeated elements cause poor performance for quicksort as well, because when partitioned, if there are no elements less than the pivot, the size of the right partition will decrease very slowly.

Quicksort Pseudocode:

```
algorithm quicksort(A, lo, hi) is
   if lo < hi then
      p := partition(A, lo, hi)
      quicksort(A, lo, p – 1)
      quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
   pivot := A[hi]
   i := lo       // place for swapping
   for j := lo to hi – 1 do
      if A[j] ≤ pivot then
         swap A[i] with A[j]
         i := i + 1
   swap A[i] with A[hi]
   return i
```

Language Choice

The five languages i've chosen are C, D, Python, Lua and Julia. This list includes fairly modern languages (Julia, D) some more mature languages (Lua, Python) and an ancient language (C).

Language	Introduced	Compiled?	Typing
C	1972	Yes	Static
D	2001	Yes	Static
Python	1991	No	Dynamic
Lua	1993	No	Dynamic
Julia	2012	No	Dynamic

Language	Compiler/Interpreter
C	gcc 5.2.1
D	DMD 2.071.0
Lua	Lua 5.2.4
Python	Python 2.7.10
Julia	Julia 0.3.11

Experiment Design

The algorithm quicksort was implemented in each language in the same way with the same choice of pivot. The pivot choice is always the leftmost value in the array partition.

The user enters a value *n* which is the number of values to be sorted. The program then generates *n* random values between 1 and 10 million and stores them in an array. The current time is recorded immediately before the execution of quicksort and immediately after it finishes. The total execution time for quicksort on *n* values is printed at the conclusion.

Prediction

I would expect the compiled languages, C and D to be the fastest. Of the interpreted languages I expect Julia to be the fastest since it was intended for scientific computing.

Results

This table shows the execution time in seconds for arrays of size *n*.

Language	*n* = 1 million	*n* = 5 million	*n* = 10 million	*n* = 25 million
C	0.32	1.79	3.72	9.47
D	0.28	1.57	3.39	8.73
C -03	0.13	0.71	1.46	3.81

optimization				
Lua	4.66	25.66	53.14	144.18
Python	4.82	34.40	67.28	189.99
Julia	6.06	53.10	160.72	841.10

Analysis

Ranked in terms of execution time from quickest to slowest we get:

1. C (-03 optimization)
2. D
3. C
4. Lua
5. Python
6. Julia

This is pretty close to what I expected. The compiled languages were faster by a big margin. Interestingly D is slightly faster than C, which I suppose can be attributed to a more modern language design. However C with compiler optimization is VERY fast, managing to reduce the execution time by two thirds. The big disappointment here is Julia. I expected that Julia would be significantly faster than Lua and Python however our results tell us otherwise, taking a whopping 14 minutes to sort 25 million values. I suspect that it has to do with Julia's garbage collector. The percentage use of the GC is shown below.

Julia	*n* = 1 million	*n* = 5 million	*n* = 10 million	*n* = 25 million
% of time spent running GC	17.08%	42.64%	56.86%	73.48%

Julia passes arrays by reference and quicksort is an in-place sorting algorithm so why is Julia using so much memory? There should be no copies of arrays being created so the insanely high GC usage is a mystery to me.

Language Critique

C:

Ease of Use:

C is a pretty easy language to use and there are no major usability problems.

Likes/Dislikes:

The main advantage of C is that it's very, very fast. This comes at the cost of static typing, which increases the lines of code in the program by quite a bit. One of the annoying things is calculating the running time, which requires dividing by CLOCKS_PER_SECOND. There should be a more user friendly way of accomplishing this.

Useful features:

C does not really have any special features.

D:

Ease of Use:

For my first time using D, I found it to be a very easy language to pick up. Despite being a multi-paradigm language that contains OO and other features, I was able to make D run the same code as C with very little modifications. This appeals to me because it doesn't force OO onto you in the way that languages like Java do.

Likes/Dislikes:

I liked that D is a more powerful language than C but is still able to run C like code with very little modifications. The main differences with C come from the way arrays are allocated, the way timing is done and the use of a built in swap function.

Useful features:

D allows for different types of arrays such as regular C-like static arrays, dynamic arrays as well as associative arrays.

Lua:

Ease of Use:

Lua is an extremely easy language to code in. In my opinion even easier than Python. The syntax is very natural and beautiful and the code is very short and concise.

Likes/Dislikes:

I liked the coding style of Lua, it has free indenting unlike Python so you don't have to match indentation if you don't want to. I find that the functions in Lua have better and more precise names than their equivalent Python functions. Lua also didn't require me to import anything for execution timing or random numbers.

Useful features:

One of the useful features that I like a lot is multiple assignment, which allows me to swap two values with a statement like this: a, b = b, a. This is quite useful in a sorting algorithm where swapping values is commonplace.

Python:

Ease of Use:

Python is an easy language to code in. Since it is a dynamically typed language, it's very fast to code in because you don't even have to consider types.

Likes/Dislikes:

One of the things that I dislike in Python is the forced indenting. I don't really think the language should force you into formatting your code in a specific way. I also don't like the way that Python does for loops, specifically the required usage of the range() function.

Useful features:

Python and Lua have a lot in common and one of them is multiple assignment which I find to be a very nice feature.

Julia:

Ease of Use:

As a dynamic, interpreted language like Lua and Python, Julia is also a very easy language to program in.

Likes/Dislikes:

Julia was a bit of a disappointment to me because it touts itself as a high performance numeric and scientific computing language, with the usability of a scripting language. The usability is there but the performance was abysmal, being by far the slowest of any of the languages that I tested.

Useful features:

Julia has a macro called @time that when placed before a function call, automatically times the execution of that function as well as memory usage and GC usage, and prints it out to the screen. Julia also supports multiple assignment.

Comparisons

Execution Timing

Language	Timing Method	Comments
C	clock_t begin,end; begin = clock(); *quicksort();* end = clock(); timeElapsed = (double)(end-begin)/CLOCKS_PER_SECOND	This method is kind of bloated, and having to cast the time value as well as divide by CLOCKS_PER_SECOND is kind of a pain.
D	StopWatch sw; sw.start(); *quicksort();* sw.stop(); timeElapsed = cast(double) sw.peek().msecs/1000;	Uses a stopwatch object rather than clock_t types in the C version. Slightly more readable to me, still requires casting however.
Lua	start = os.clock()	Short and elegant, very

	quicksort() timeElapsed = os.clock() - start	readable.
Python	start = time.time() *quicksort()* elapsedTime = time.time() - start	Nearly identical as Luas style.
Julia	@time *quicksort()*	By the far the best timing mechanism i've seen in any language. You only have to add @time in before any function call and it tells you timing, memory usage as well as percentage of time spent using the GC.

Array Initialization

Language	Initialization Method	Comments
C	int A = malloc(sizeof(int)*size);	This way is precise and allows lots of control for the programmer but is longest code of any tested.
D	int A[]; A.length = size;	Much simpler than the C style, very easy to understand. Allocates a dynamic array similar to what C does.
Lua	A = {}	Creates an empty table, Lua's only data structure. Size is dynamic and increases as more elements are added to it.
Python	A = []	Creates an empty list. Size is also dynamic and increases as more elements are added to the table

Julia	A = cell(size)	Have to give the array size during initialization. Creates a simple array in the dimensions of size.