Clark University
Clark Digital Commons

Academic Spree Day and Fall Fest | Academic Spree Day 2020

May 17th, 12:00 AM - 11:00 PM

Lobster Programming Language

Skyler Brivic
Clark University, SkBrivic@clarku.edu

Follow this and additional works at: https://commons.clarku.edu/asdff

Brivic, Skyler, "Lobster Programming Language" (2020). *Academic Spree Day and Fall Fest*. 5.
https://commons.clarku.edu/asdff/ASD2020/Posters/5

This Open Access Event is brought to you for free and open access by the Conference Proceedings at Clark Digital Commons. It has been accepted for inclusion in Academic Spree Day and Fall Fest by an authorized administrator of Clark Digital Commons. For more information, please contact mkrikonis@clarku.edu, jodolan@clarku.edu, dlutz@clarku.edu.

Lobster Programming Language

Skyler Brivic '20 – (Sponsor: Professor Frederic Green)

Main Purpose:

Most programming languages treat computers as single processor entities, even though the vast majority of computers produced today have 2 or more processors. Imagine driving a car that used only half its engine! Such is the fate of most programming languages… But what if a language could be designed with multi-threading as a core component of its design? That is where the new programming language that I've developed comes into play: Lobster. As the name suggests, this language is intended to be resilient, hard to crack, and to be able to last for a long time!

Language Structure:

Lobster code is syntactically very similar to Java or C++. In order to create a running program, the code typed in by a user must first be compiled into a special bytecode, which can then be run on the Lobster Virtual Machine (the language is an interpreted programming language).

First, a preprocessor reads through the code written by the user, and combines files with include statements appropriately. Then, a parser uses a lexical analyzer to extract tokens from the file and build complete parse trees and symbol tables for the program. The parse tree and symbol table are then fed as input into the code generator, which outputs a bytecode representation of this code that can be interpreted by the Lobster Virtual Machine

Each instruction in Lobster is represented as a 15-byte sequence, where the first byte contains the Opcode that specifies what type of instruction it is. If the instruction takes up less than 15 bytes, then the remainder of the 15 bytes is padded with 0s.

Built-in Types:

In Lobster, there are 3 built-in classes: Arrays, Lists and Queues. Each structure makes use of locks to ensure that multiple threads can access an object of the built-in class without causing any data races. Queues use a version of the Michael and Scott Queue, which allows for simultaneous insertion and deletion into a queue without causing data races by initializing the queue with an empty dummy node and using 2 locks to control the head and tail ends of the queue. Both Arrays and Lists use one big lock which is acquired on entry to each built-in function and released at the end of each built-in function.

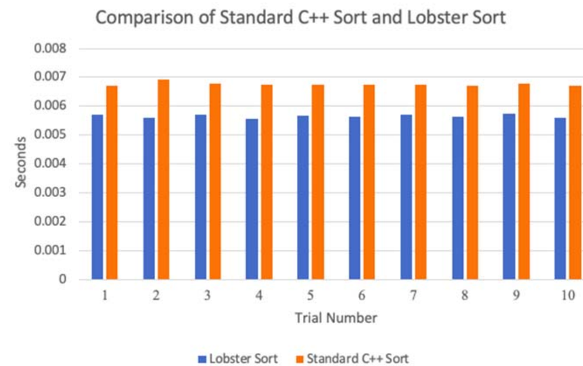

Fig 1: Results of sorting a randomized array of 100,000 integers using the C++ standard sort and using my Lobster Sort. Note that each trial consists of sorting 1,000 arrays, with the average time per sort recorded as the result for each trial.

Lobster Sort

In Lobster arrays, there is a built-in sort function I have written, which I have named Lobster Sort. This sort is a variant of mergesort/timsort which proceeds as follows: create a second thread, and then call the sort helper method on each half of the array. In the sort helper method, if the size of the subarray that is about to be sorted is less than 60, then insertion sort is performed on the array. Otherwise, the sort helper method is recursively called again on the first and second halves of the array. Once each half is sorted, the merge algorithm used by merge sort is used to combine each half into a fully sorted array. Once both threads finish, the merge algorithm is used on the two halves of the array to produce one final, fully sorted array. This sorting algorithm is stable, has a space complexity of O(N), and has a time complexity of O(N * LogN).

As can be seen from the results of the tests displayed in fig. 1, Lobster Sort was able to sort an array of 100,000 random integers at an average time which was approximately 20 % shorter than the average time it took the C++ standard sort to sort an array of 100,000 random integers. Through the power of multi-threading, massive time-saves are possible!

Future Development

One feature which I am planning to add to Lobster is pipeline structures. In a pipeline, several functions will work simultaneously in separate threads, with the input of one function being sent to a queue which acts as input to the next function in the pipeline. A function in the pipeline will dequeue an entry from its input queue, process the value, and then jump back to the beginning of the function when it's done to wait for more input. When a function in the pipeline pushes a TERMINATE signal onto an input queue, the thread containing the function will end, and any function in the pipeline which dequeues the TERMINATE symbol will push a TERMINATE symbol onto the next input queue, and will finish execution as well. This structure has the potential to save a lot of time in cases where many computationally intensive calculations need to be done in parallel, where the input of one calculation is only needed at a small number of points in later calculations in the list.

I would also like to include an Event-Dispatch structure, in which one central thread would continually push arguments onto the input queues for many other functions, and all of those functions would execute in parallel in separate threads until the central thread sends out a TERMINATE signal to each thread.

```
void sort()
{
        if(size == 0)
                return;

        if(size < SINGLE_THREAD_LIMIT)
        {
                writeLock->lock();
                LobsterSort(0, size - 1);
                writeLock->unlock();
                return;
        }

        else
        {
                writeLock->lock();
                std::thread otherThread(&UserArray::LobsterSort, this, 0, size / 2);
                LobsterSort(size/2 + 1, size - 1);
                otherThread.join();
                merge(0, size / 2, size - 1);
                writeLock->unlock();
                return;
        }

}
```

Fig 2: The sort code for built-in arrays in Lobster. If the array is smaller than 300 elements, then a single-thread merge sort is done . Otherwise, two threads sort each half of the array simultaneously, and merge their subarrays into a final sorted array when finished. It is this sorting algorithm which I have named Lobster Sort.