# Literature Review

Paper 1: Practical Distributed Programming in C++

Paper 2: On the Validated Usage of the C++ Standard Template Library

Paper 3: Relaxing the One Definition Rule in Interpreted C++

By- Sarthak Dalmia

ID- 2018B3A70290G

## Paper 1: Practical Distributed Programming in C++

At high levels of abstraction, the C++ language lacks support for distributed memory systems. The Standard Template Library (STL) includes containers and algorithms as primary notions, coupled with execution policies that allow exploiting parallel platforms (e.g., multi-cores) on top of a well-defined operational semantics. This paper tackles the above problem to some extent. This discusses the design of a stack for STL compliant containers, iterators, algorithms, and execution policies targeting distributed-memory systems. Finally, the above approach is evaluated by analyzing the performance of the proof-of-concept approach over a set of STL algorithms.

A distributed computer system consists of multiple software components that are on multiple computers, but run as a single system. The computers that are in a distributed system can be physically close together and connected by a local network, or they can be geographically distant and connected by a wide area network. This system provides scalability and redundancy.

This work proposes a stack whose bottom layer provides distributed tasking and serves as lightweight support for the partitioned data structures layer. Both layers are included in the Scalable High-Performance Algorithms and Data-structures (SHAD) library, that is exploited as foundation. A distributed STL container is defined to have the same API as a regular STL container, and therefore a C++ program using STL can be ported to distributed STL by mere namespace substitution(dstd). Currently only contiguous (array) and associative containers (unordered_set and unordered_map) are provided but it can be easily expanded to all containers. Changing std to dstd for a container changes all its iterators and ranges to distributed form. Distributed iterators have the same semantics as normal iterators.

Local iterators had to be introduced to avoid the overhead of locating the locality where the actual value is present in direct dereferencing. They are STL compliant and gives direct access to underlying data. The distributed allows referencing in a lazy manner, which means if the reference occurs as left side operand or right-side operand there will be remote access of write or read are triggered respectively. Optimizations are placed to reduce the overhead caused due to the remote access. For contiguous containers iterator arithmetic works as usual but for associative containers remote task might be triggered to the locality which has the next item.

Regarding of distributed algorithms, naively passing distributed ranges to STL algorithms would result in performance penalties (due to remote read-write dereferencing). To tackle this new execution policies have been created namely distributed-parallel, and distributed-sequential to give new semantics to distributed algorithms which work in parallel and sequential mode respectively. Most algorithms access single range (to read elements or modify them in place), which allows avoiding any remote access under both the policies. Multi-range algorithms have no guarantee that the sub-ranges accessed belong to the same locality thus there is remote access. To combat this, three optimizations are done for the cases: accessing distributed iterators over node-local sub-ranges, remote output operations over physically contiguous memory blocks like arrays, and to mitigate latency even when no assumption can be made about data layout(maps).

Finally, upon evaluating the above approach, performance of distributed single range algorithms is comparable to their STL counterparts. As multirange algorithms have remote access problem though they are slower than STL counterparts, the optimizations do seem to work effectively by significantly reducing the overhead if not fully eliminating it (in some cases).

## Paper 2: On the Validated Usage of the C++ Standard Template Library

The C++ Standard Template library (STL) is the most well-known and widely used library that is based is based on generic programming paradigm. After so many years of development on the same, still a large number of properties are neither tested at compile time nor at runtime. This issue can sometimes create new problems for the programmer to deal with. This paper gives four different approaches to deal with the above problem.

The first approach is metaprogramming. This approach is based on the template construct of C++ which is able to evaluate Turing-complete checks at compilation time. One can modify the STL implementation to evaluate specific instantiations in this approach. Also using the metadata collected specific cases can also be checked. This does require the modification of the STL implementation though.

Second approach is also a compile time approach. In static analysis, pattern matching is performed on the AST generated during the compilation of the programming along with compilation arguments. The matched nodes are collected and then checked for exclude lists to remove any false positives. Lastly all the remaining matches are translated to user-friendly warning messages.

The third approach is a debugger-based runtime validation approach. This approach is based on the gdb debugger tool. Breakpoints are setup on specific locations in the library and when the execution is suspended the intervals, iterators, the state of the containers, etc. are evaluated. After evaluation, one can continue the execution or indicate an error. Correct and convenient usage of gdb requires compilation with debug mode enabled. But as there are breakpoints significant runtime overhead is there.

The last approach consists of aspect-oriented programming. The main idea behind this technique is to extend or modify the original program runtime behavior without changing the source code. It usually is done by adding extra source code just before the compilation. This extra source code is woven into original source code, but it is not changed physically. Thus extra check conditions for errors can be added and compiled using first the AspectC++ to convert to valid C++ code then proceeding with normal compilation and execution.

The above four approaches are compared on various factors. In sophistication, (how many problems can be addressed) of course the runtime approaches are superior overall, with debugger-based approach beating aspect oriented one and static analysis beating metaprogramming. All approaches except metaprogramming are non-intrusive. Metaprogramming and gdb approach require specific STL implementation whereas the rest two approaches are implementation free. Same goes for applicability. Libraries are not supported for obvious reasons in the gdb based and metaprogramming approaches but opposite is true for the rest of the approaches. When it comes to overall performance metaprogramming is best followed by static analysis then aspect-oriented solution and then debug mode solution., as they both have runtime overhead. Debug mode is the easiest to implement where as the aspect-oriented approach is most modular.

Finally the paper suggest the use of runtime approaches during testing phase because they detect more problems and gives less false positives then compile time approaches. Overall static analysis can be improved in further studies.

## Paper 3: Relaxing the One Definition Rule in Interpreted C++

Interpreted execution of C++ have significant use cases in scientific fields and such is demonstrated by the Cling interpreter from the ROOT project at CERN. Thus it is desirable to make this implementation as useful as possible. One step in doing so is relaxing the One Definition Rule of C++ which restricts users to redefine functions, variables and classes with same name at a later point in the program. This paper tackles the above problem by modifying the Cling interpreter to allow redefinition of declarations by nesting each declaration into a nested namespace and thereby shadowing (but not deleting) the previous declaration of the same name.

Cling interpreter is used by people expecting Python-like interaction. Cling first checks whether an input is required to be wrapped in a function. This is due to the reason that raw input lines are not directly supported by C++. Thus, after wrapping any invalid statements the code is parsed using the usual Clang infrastructure. This gives an AST as the output for the parsed top-level declarations. These are added to the translation unit declaration list. This is where the relaxing of ODR takes place. AST transformers can be created to support various Cling features. Lastly, the LLVM is used for just-in-time compilation and execution.

To understand the transformation suggested by the paper, one must understand a few basic concepts. Namespaces are a declarative region that provides a scope to identifiers (the names of types, functions, variables, etc.). All the identifiers in the namespace are visible to one another without qualification (that is without specifically specifying the namespace along with the name). Identifiers outside can access the members via 3 methods, namely, using fully qualified name for each identifier for e.g. std::vector<std::string> vec;, or else by a using Declaration for a single identifier (using std::string), or a using Directive for all the identifiers in the namespace (using namespace std;). Ordinary nested namespace have unqualified access to parents identifiers but opposite is true for the parent namespace. Inline namespace members on the other hand are treated as member of parent namespace that is they are present in both the parents and the namespace's itself lookup table.

In this paper declarations are nested into new inline namespaces (say DefinitionShadowerNS). Exception are the identifiers declared using the using-directive or using-declaration (as they already TU already have unqualified access to those). By doing so each declaration definition is present in both the DefinitionShadowerNS declaration list and the TU declaration list. Problem arises when multiple inline namespaces have definitions for same name. This is rectified by a simple tweak to the TU lookup table. As soon as a defined declaration comes the AST Transformer deletes all the entries in TU lookup table corresponding to the same name except in the case they are overloaded. Next, as enumerators declared in unscoped enumeration are visible in parent context(TU here) they also must be removed from the lookup table. Lastly, all the non-definition declarations after the input are also ignored. A point to note is that, even though names were deleted from the TU lookup table, these declarations are still accessible through there fully qualified names.

A few limitations of the solution were that the shadowed global objects were though invisible from TU lookup table but the memory it was referencing was still allocated. Another limitation was that as many qualified name of types changes run-time type identification becomes a problem. Lastly the paper validated the approach using extensive set of examples. For the sake of completeness overhead was also calculated with shadowing enabled which ranged from 4-52% in function and class redefinitions and 2-13% in variable redefinitions as compared to no shadowing.

## My Research and Study

As this field of computer science is huge, there was a large plethora of papers to choose from. I decided to go for papers with C++ related optimizations and improvements. After studying so many papers, research articles and websites related to above topics, I have gained a whole new level of knowledge in the field of programming.

The concept of multi-threading and parallel computing is a big area in the field which according to me can be further improved upon by applying distributed systems onto it. The paper I studied and researched for this assignment had a few limitations. It was not implemented for all the containers in the STL library. Further work to eliminate the above problem can be done. As the approach complied with the standard STL implementation, it removes the need to learn whole new concepts and methods to make use of distributed systems.

STL library is used all over the world by millions of people. As the library is frequently updated with new standards, we also need to improve the error detection methodologies used in the same. The approaches suggested by the paper were quite good. Though the metaprogramming approach was the fastest, it was not sophisticated enough to address significant number of problems as it could only the used on code that was actually run. Aspect oriented programming needed a special aspect weaver to be implemented. An aspect weaver is designed to take instructions specified by aspects and generate the final valid code for implementation, in our case C++ code. The debugger approach was indeed the easiest to implement but also required the largest overhead during run-time. The static analysis approach was, according to me, the most viable approach as it was non-intrusive, implementation free, suite sophisticated when compared to metaprogramming with room for improvement. An approach suggested over other research papers was using the symbolic execution engine to improve the static analysis to include runtime reasoning. A symbolic execution engine will typically represent the program's memory in an engine-specific data structure and, when the execution reaches a branch whose condition in-volves symbolic values, the engine forks the program state, such that each path has its private version of the program state.

Lastly the relaxing ODR paper was the most informative for me. To understand the paper, I had to do quite an in-depth research on namespaces and templates. Moreover, I had to learn how interpreted languages like Python works. The method suggested is quite ingenious. The paper uses inline namespaces to effectively hide but not completely delete declaration from the translation unit. The paper has listed some of the limitations of the approach which they have claimed will be addressed in future work. We can combine multiple domains. The interpreted C++ language implemented here still requires error detection, thus techniques like debugging can be implemented to the JITed code to get better reliability and lesser problems. The relaxed ODR Cling is currently at use in the CERN’s ROOT Project by domain experts in fields like high Energy Physics, etc.