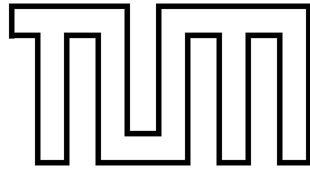


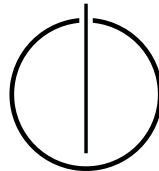
DEPARTMENT OF INFORMATICS

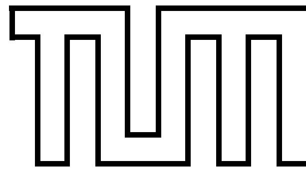
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Enabling the compilation of LAMMPS on the invasive Run-time Support System

Lukas Kirchmair





DEPARTMENT OF INFORMATICS

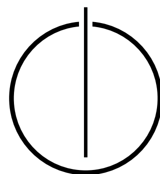
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Enabling the compilation of LAMMPS on the invasive Run-time Support System

Ermöglichung der Kompilierung von LAMMPS auf dem invasiven Laufzeitunterstützungssystem

Author:	Lukas Kirchmair
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Santiago Narvaezl, M.Sc.
Date:	15.05.2022



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.05.2022 Lukas Kirchmair

Acknowledgements

This thesis would not have been possible without the support of many people. Many thanks to my adviser, Santiago Narvaezl, who read my revisions and helped make some sense of the confusion.

Thanks to my girlfriend who endured this long process with me, always offering support and love.

Finally, I must express my very profound gratitude to my parents and to my sister for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of this thesis. This accomplishment would not have been possible without them.

Thank you.

Abstract

Standard Template Library (STL) is a universally used set of C++ template classes. The STL provides the most commonly used programming data structures and functions, such as lists, stacks, and arrays. All STL container classes, algorithms, iterators, and other elements are parameterized to be generalized. The library has many essential and highly optimized elements that almost every modern C++ code uses it. This also includes programs that run on new operating systems like OctoPOS, designed from the ground up for future many-core architectures. In the case of the molecular dynamics simulation LAMMPS in the combination of OctoPOS, an STL implementation is missing. To be more specific, the LAMMPS source code depends on the MPI-API, which OctoPOS already provide, and a C++11 STL implementation is missing. This thesis explains what the Standard Template Library is in detail and where the library is defined. Furthermore, how to implement them and test core elements from OctoPOS.

Zusammenfassung

Die Standard Template Library (STL) ist eine universell verwendete Sammlung von C++ Vorlagenklassen. Die STL bietet die am häufigsten verwendeten Programmierdatenstrukturen und -funktionen, wie lists, stacks und arrays. Alle STL-Containerklassen, Algorithmen, Iteratoren und andere Elemente sind parametrisiert und können verallgemeinert werden. Die Bibliothek enthält so viele grundlegende und hoch optimierte Elemente, dass fast jeder moderne C++-Code darauf aufbaut. Dazu gehören auch Programme, die auf neuen Betriebssystemen wie OctoPOS laufen, das von Grund auf für zukünftige Many-Core-Architekturen entwickelt wurde. Im Falle der Molekulardynamiksimulation LAMMPS in der Kombination mit OctoPOS fehlt eine STL-Implementierung. Genauer gesagt, hängt der LAMMPS-Quellcode von der MPI-API ab, die OctoPOS bereits zur Verfügung stellt, allerdings fehlt eine C++11 STL Implementierung. Diese Arbeit erklärt, was die Standard Template Library im Detail ist und wo diese definiert wurde. Außerdem wird erklärt, wie man sie implementiert und die Kernelemente von OctoPOS testet.

Contents

Acknowledgements vii

Abstract ix

Zusammenfassung xi

I. Introduction and Theory/Background 1

1. Introduction 2

2. Related work / Background 4
- 2.1. LAMMPS 4
- 2.2. Invasive Computing 4
  - 2.2.1. Invasive Operations 5
- 2.3. OctoPOS 6
  - 2.3.1. OctoPOS Design and Goals 6
  - 2.3.2. OctoPOS Hardware 7
- 2.4. Standard Template Library 8
  - 2.4.1. Structure of STL 8
  - 2.4.2. History of STL 10
  - 2.4.3. C++ Committee 10
  - 2.4.4. ISO IEC 14882 11

II. Thesis Development 12

3. The requirements to make LAMMPS compile 13
- 3.1. LAMMPS build 13
  - 3.1.1. CMake 13
  - 3.1.2. make 13
- 3.2. LAMMPS dependencies 14

4. Implementation of the libraries 16
- 4.1. Implementation process 16
- 4.2. Implementation examples 16
  - 4.2.1. stable partition 16
  - 4.2.2. map 18
- 4.3. Library dependencies 21

4.4. Implementation references 22

5. Testing framework **24**
5.1. Catch2 testing framework 24
5.1.1. Why use this testing framework? 24
5.1.2. Why do we use Catch2? 25
5.1.3. Why write unit tests 25
5.2. Testing in OctoPOS 26
5.2.1. OctoPOS lib test concept 26
5.2.2. Test example implementation 26

III. Results, Conclusions and Future work **28**

6. Comparison of efficiency between OctoPOS and normal Linux **29**
6.1. stable partition 29
6.2. map insert 30
6.3. x86guest vs x64native 31

7. Testing **33**
7.1. Tests build setup 33
7.2. Test execution 34
7.3. Shortcomings 34

8. Conclusion **35**

9. Future work **36**
9.1. Performance improvements 36
9.2. Library extensions 36
9.3. Exception handling 36
9.4. Using libstdc++ 37

IV. Appendix **38**

A. LAMMPS dependencie list **39**

B. Complete libc++ dependency tree **40**

C. OctoPOS tests directory tree **41**

Bibliography **45**

Part I.

Introduction and Theory/Background

1. Introduction

We nearly reached a time when multi-core architectures were getting out of fashion. Not long ago, everything started with single-core CPUs. Intel released its Intel 4004 in 1971, with it they revolutionized the microprocessor design and the entire integrated circuit industry with it [Bio]. The next big step for Intel was from the 4004 to the 8086, which had ten times more transistors. Furthermore, every new CPU generation had more performance by combining an exponential increase of transistors and the addition of dedicated floating-point units, multipliers, and improvements in the general instruction set and its extensions. In those times, technologies and designs that nowadays we take granted, were just being developed, like the pipeline CPU design, which started with the Intel i386. Nevertheless, the most significant contributor to the immense CPU performance increase was "simply" decreasing the transistor size, so it was possible to put more transistors in the same space as before. In early 2000, the CPU manufacturers increased their clock speeds, new CPU designs were created and optimized, and new features like branch prediction and multi-threading were introduced. These innovations peaked in the top single-core CPU of its time, the Intel Pentium 4 with 3.8GHz and support for two threads.

Everything indicated that at some point, we would have a CPU that would run at 10GHz. However, the increasing clock frequency and shrinking transistor sizes resulted in higher power consumption, increasing the risk of current leaking. Moreover, all that current running through the CPU produces much heat that is not efficiently dissipated. All these problems meant there was a ceiling for the CPU frequency for consumers. Therefore a new CPU design was needed, so multiple single-core processors were connected to one big multi-core processor.

Nowadays, a system consists of multiple physical CPUs that share one primary memory pool and peripherals on one single motherboard. Only super-computers and servers used the first such systems, and the first actual multi-core processor got released by IBM under their Power4 architecture in 2001. Over time the GHz race slowed down, and regular consumers used the multi-core processor systems. These days AMD and Intel are selling consumer multi-core CPUs, where one CPU can have up to 16 cores and 32 threads.

After multi-core CPUs, the following step for super-computer and server CPUs might be a many-core architecture with 10^3 and more processors on a single chip [OSK+11]. The shared and distributed memory could coexist on a single chip on these CPUs. We might have so many cores on a single many-core system that every thread will run on its private core at some point in the future. At that point, multi-threaded cores will be the exception, and many-core systems will be the default. When we reached that point in hardware, we also needed a new way for operating systems to manage processors, and programmers needed a way to manage fine-grained parallelism.

A way to benefit from these new multi-processor systems on chip (MPSoC) got introduced in "Invasive Computing: an Overview" [THH+11]. From that emerged OctoPOS, a redesigned from ground up OS developed from the Transregional Collaborative Research

Centre “Invasive Computing” (SFB/TR 89) funded by the German Research Foundation.[1] With OctoPOS, a new OS got developed that is optimized for the future many-core architecture that renounced heavyweight threads. In addition, OctoPOS includes a partial implementation of MPI. That way, it is possible to run existing C-MPI programs without any modifications.

A problem arises when we want to compile a C++ MPI program, like the molecular dynamics simulation code LAMMPS, which is written in C++11. OctoPOS is missing an implementation of the Standard Template Library (STL). Thus, this thesis aims to implement a part of the STL special fitted to OctoPOS's current limitations. In addition, the thesis tries to provide answers to the following questions:

- Which dependencies does LAMMPS have, and how to find them?
- Where to find the STL definitions?
- How to test the OctoPOS components? (including the special version of the STL)
- What are the performance differences between the existing STL and the OctoPOS STL implementation?
- What are the performance differences between the special STL on bare-metal and a guest system?
- Is it worth implementing and having a special version of STL?

[1] `http://www.invasic.de`

2. Related work / Background

2.1. LAMMPS

LAMMPS stands for **L**arge-scale **A**tomic/**M**olecular **M**assively **P**arallel **S**imulator [TAB+22, LAM21]. As the name already indicates, LAMMPS is a molecular dynamics simulation code with a focus on materials modeling. It is specially designed for parallel execution. The LAMMPS project is open-source, and it is distributed freely under the terms of the GNU Public License Version 2 (GPLv2). Under the GPLv2 license, LAMMPS can be used free of charge, and it can be modified for different purposes, but there are specific rules if the changes redistribution is wanted.

LAMMPS can simulate particles in liquid, solid, or gaseous states. The LAMMPS documentation lists every system that the simulator can model, among others, atomic, biological, coarse-grained, granular, macroscopic, polymeric, or solid-state (metals, ceramics, oxides). In addition, LAMMPS uses a variety of interatomic potentials (force fields) and boundary conditions. The simulation can be in 2D or 3D systems with only a few particles or up to millions or billions of them.

LAMMPS is designed and optimized for parallel computers, but it can be built and executed on consumer laptops or desktop machines. By default, LAMMPS runs in a serial process. The MPI message-passing library is used to utilize the full potential of the parallel systems. Some parts of LAMMPS also use OpenMP multi-threading, vectorization, and GPU acceleration. Spatial decomposition combined with MPI parallelization divides the simulation domain into smaller chunks of equal computational cost to distribute the load on parallel machines.

The codebase of LAMMPS is written in C++ and requires a compiler that is at least compatible with the C++-11 standard. The code structure is very modular, so it is straightforward to modify and extend LAMMPS with new features. Newton's equations of motion are integrated into a collection of interacting particles at the core of LAMMPS. Depending on the simulation scenario, a particle can be an atom or a molecule, or an electron, a coarse-grained cluster of atoms, or a mesoscopic or microscopic clump of material. A neighbor list is used to keep track of particles that can interact with each other. The lists are optimized for systems with repulsive particles at short distances so that the local density of particles never becomes too large.

2.2. Invasive Computing

The idea behind Invasive Computing and Programming is defined in [THH+11] as following:

Invasive Programming denotes the capability of a program running on a parallel computer to request and temporarily claim processor, communication and memory resources in the neighborhood of its actual computing environment, to then execute in parallel the given

program using these claimed resources, and to be capable to subsequently free these resources again.

With these definitions, the authors of [THH+11] hope to be able to benefit from future multiprocessor system-on-chip (MPSoC) by allowing programs to manage and control the processing resources (i.e., processor cores, communication links, cache lines), but only up to a certain degree and in context of the local state of the underlying computing hardware.

2.2.1. Invasive Operations

A particular programming paradigm and notation are needed to enable invasive programs to request and control processing resources. After claiming the resources, the operating system decides how many of them the program can use. Because of that, an invasive computing program should be able to adapt and be aware of its resources. The program might change its behavior in the case it did not get all the wanted resources based on the three basic primitives, which are described in [THH+11] as follows:

invade A request indicates that an entity needs more hardware resources. If the invasion is successful, the run-time support system will assign the requested resources to a claim.

infect The command will infect a granted resource and its process with the program, much like a virus. After successful infection, the parallel execution may start on all infected resources.

retreat The retreat command leads to freeing the claimed resources by the run-time systems if the retreat was successful.

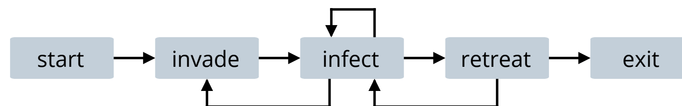


Figure 2.1.: State diagram of an invasive program [OSK+11]

The typical state transitions of an invasive program is shown in Figure 2.1. The live cycle starts with the construction of the initial claim. The claim construction is done by issuing a call to invade. If the invasion was successful, a program would be granted exclusive access to some resources. At the same time, it is not required to have a mechanism for resource-sharing because there is a superior instance that decides how the resources are distributed. After a successful invasion, infect is used to start the actual application code on the claimed resources. As soon as the execution finishes on all resources, the number of resources inside the claim can be altered. That can be done by calling invade or retreat to expand or shrink the application's claim. If there is no change in the degree of parallelism, it is also possible to dispatch another program onto the same resources by another call to infect. A program needs to retreat from all resources by leaving the claim empty to exit. Therefore there are no computing resources left for further execution. Hence it terminates its execution.

In [THH+11] a basic unit of execution, i.e., a piece of program subject to invasive-parallel execution, is referred to as an i-let, which is short for "invasive-let". An i-let got defined as a section of a program that will potentially be executed in parallel. If only one process unit was invaded, or because of other scheduling decisions, it may be executed in serial. A group of i-lets is called a team. Like the desired scheduling technique, hints about run-time behavior can be added to a team. After its creation, a team is used to infect a given claim.

2.3. OctoPOS

OctoPOS is an operating system designed and implemented specifically for the Invasive Computing paradigm to address the hardware and software challenges of future massively parallel systems like a Multi-Processor System on a Chip (MPSoC). The name OctoPOS is a combination of the prefix 'Octo' and the abbreviation 'POS', which stands for a parallel operating system. The prefix 'Octo' refers to the octopus, which can use all its tentacles in parallel and adapt very well to its environment. The properties mentioned of parallelism and adaptability are also of central importance for OctoPOS.

2.3.1. OctoPOS Design and Goals

OctoPOS was designed as an Invasic Run-time Support System (iRTSS), which, together with the agent system, provides the basis for the execution environment. In particular, OctoPOS is designed explicitly for a Partitioned Global Address Space (PGAS) programming model like it is used with tiled many-core architectures [MBZ+15]. The cores of such architectures see a global address space, but cache coherence is only guaranteed for a group of cores that form a tile. Inter-tile communication is done over a simple scaleable interconnection in a message-based fashion. Not relying on global cache coherency allows OctoPOS to support large systems efficiently.

Several design goals were pursued in the design of OctoPOS. These are briefly listed below:

Resource-aware operating system Resource management must be extended for resource-aware programming. In addition to managing existing resources, exclusively reserved resources must be considered when allocating them. The abstractions within the operating system must themselves be resource-aware.

Micro parallelism Efficient use of the available processors requires lightweight support for application parallelism. Even small programs should be able to be executed in parallel with the help of efficient abstractions. For these abstractions carriers are required to have low costs and the critical path as short as possible for the activation and loading of activity.

Scalability The operating system should scale well and not become a bottleneck in executing parallel applications.

Availability for application development Simultaneous design and implementation of the hardware and software requires an alternative platform for independent development of the software. Therefore, OctoPOS will be designed and maintained for multiple

architectures. This also decouples hardware and software development, which speeds up development and can also be used for debugging.

Hardware support The hardware extensions available on the architecture for "Invasive Computing" are used efficiently by the operating system. In doing so, classical tasks of an operating system - if available - are taken over by hardware components. However, software emulations are also required for architectures on which invasive hardware components are unavailable.

2.3.2. OctoPOS Hardware

The Parallel execution implementation from OctoPOS is very lightweight compared to other comparable operating systems. Switching overhead between two i-lets after the first one is finished is similar to the overhead of calling an indirect function [OSK+11].

An i-let is a structure with a void-function pointer and another pointer to an additional piece of data. When OctoPOS executes an i-let, the function pointer will be invoked with the data pointer as its only argument.

The OctoPOS version used runs on computing cores accessing the same piece of memory with uniform access costs. OctoPOS supports neither spatial nor temporal preemption to support a slimmed-down operation system [OSK+11].

As a result, an executed i-let will run till its completion unless the i-let itself calls a blocking system call. In that case, OctoPOS will stop the blocking execution and remove the i-let from the processing unit. At some point, the blocking system call will end, and the i-let will want to return to its execution. For that reason, the execution context needs to be temporally saved. Similar to other traditional operating systems, OctoPOS as well as a concept of process context that is typically represented by a stack on which an i-let is executed. At the current state of the implementation, for OctoPOS, a context consists of a stack frame, a field for a saved stack pointer to resume the context later on, and a field indicating if the context is used or free. A blocked i-lets instruction pointer and its non-scratch registers are getting pushed on the current context's stack. The stack pointer will be saved, and the context itself will be removed from the processing unit to make space for the next one. Out of the pool of new contexts, one will be selected and fetched to start the execution on the available processing unit. After the blocking system call ends, the i-let can be rescheduled. The new i-let will be exchanged by the no longer blocked one, and the new i-lets context will be moved back into the pool of free contexts.

Three targets are supported by OctoPOS [OSK+11]. These are:

SPARC Leon The initial architecture used for invasive computing. It features a hardware-based Core i-let Controller (CiC), which is responsible for dispatching ready i-lets to the available cores of the claim.

x86 Guest Compiling OctoPOS for this target produces an x86 binary that can be run under Linux. If executed, OctoPOS runs as a "guest" program under the Linux operating system. This target is not intended to be used in production, only for development and debugging.

x64 Native Targets standard x64 (x86-64, AMD64) hardware. On platforms with multiple Nonuniform Memory Access (NUMA) domains, OctoPOS creates a tile for every NUMA domain.

OctoPOS is ideal for large and highly parallel programs that take advantage of microparallelism, mainly through the fast and scalable execution model and the provided lightweight primitives for executing i-lets and efficient synchronization mechanisms.

2.4. Standard Template Library

The C++ Standard Library is a collection of classes and functions in the C++ programming language, and it is part of the C++ ISO Standard (ISO/IEC 14882) [2211]. The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, and arrays. The library also includes often used container classes, algorithms, and iterators. Moreover, the STL gets its name from being a generalized library, so its components are parameterized. The benefit of using the STL is that it saves time and effort by providing reliable and fast data structures and algorithms, which can be easily extended.

2.4.1. Structure of STL

The STL can be divided up into the following main categories, with those in bold being relevant to this thesis:

- **Algorithms**
- Atomic operations
- **C library wrappers**
- Concepts
- **Containers**
  - **Sequence containers**
  - **Ordered associative containers**
  - **Unordered associative containers**
  - Container adapters
- Container views
- **Error and exception handling**
- **General utilities**
- I/O and formatting
- **Iterators**
- Language support
- Localization
- **Math and numerics**
- **Memory management**
- Multi-threading
- Ranges
- Regular expressions
- **Strings and character data**
- Time

The categories, which are important for this thesis, are described in the following subsections [Mic]:

Algorithms

Algorithms are a fundamental part of the C++ Standard Library. Algorithms mainly use iterators and do not work with containers. Therefore, all container classes provide a custom iterator.

The three headers <algorithm>, <cstdlib>, <numeric> belong to the category of the algorithm.

C library wrappers

As the category name hints, 26 header files wrap C functionality inside the std namespace. However, sometimes C functions and macros are undefined and re-implemented in the std namespace. Examples for such are wrapper are: <cassert>, <cmath>, <cstdio>, which wrap the <assert.h>, <math.h>, <stdio.h> respectively.

Containers

STL defines a wide range of types-safe containers to store and manage a wide variety of objects. The type of elements that the containers are storing is defined over templates at compile time. All containers can be constructed with a `std::initializer_list`, and they all have member functions to add, remove elements, and do other operations. We move between the container's elements and access them individually with iterators. The iterators are either used explicitly by calling the member functions or implicit by range-for loops. All C++ STL iterators have a standardized interface, and all containers have their specialized iterators. For example the `std::map` class has a tree iterator.

The container category can be divided into three subcategories: sequence containers, associative containers, and container adapters:

Sequence Containers Sequence containers maintain the ordering of inserted elements that you specify. Typical sequence containers are `std::array` and `std::vector`, which have a fixed and dynamic length respectively

Associative Containers In contrast to other containers, Associative Containers store "sorted data" compared to other containers. The benefit of this is, searching through ordered data much faster than random ordered ones. The downside is that the inserting of new elements takes much longer. The associative containers can be grouped into two subsets: maps and sets. Maps consist of key/value pairs, and every key is unique. A set only allows one instance of a key, and the value is also the key. The keys are used to order the inserted elements in both maps and sets.

General utilities

In this category, we have general useful structs, functions, and operators, like inside the <utility> header, where the struct pair is defined, which is used when two objects need to be treated as one. Alternatively, a tuple can hold an object of varying types from the same-named <tuple> header. Inside the <type_traits> header, templates for compile-time constants are defined, giving information about their type arguments' preterites or producing transformed types.

Iterator

The iterator category only has one element: the <iterator> header file, which defines iterator primitives, predefined iterators, and stream iterators. STL has some predefined specialized iterators, like the `std::reverse_iterator`, which uses another iterator and reverses its operators.

Memory management

The memory management includes the <allocators>, which defines several templates that help allocate and free memory blocks for node-based containers. Furthermore, the <new> header controls allocation and freeing storage with types and functions and defines components for reporting storage management errors.

Strings and character data

In this category, the basic_string class is defined inside the <string>, which is the basis for the std::string alias. In addition, operations and comparisons of char and double-wide char arrays are defined or wrapped inside the std namespace.

2.4.2. History of STL

Alexander Stepanov was the first who suggested the basic design and idea behind a standardized template library [Shu]. He started working on his initial idea of generic programming in 1979 and explored its potential for a revolution in software development. In 1971, David Muser had already developed and advocated some aspects of generic programming. It was limited to a rather specialized area of software development.

ADA was the first programming language used to generate a generic library. Stepanov and Musser published the ADA library in 1987, with generic list processing as the first implemented component of the library.

Stepanov switched from ADA to C++ for its fast access and managing of data storage due to pointers. The crucial point was to achieve a generality without losing efficiency, which was possible by having flexible access to storage via pointers. Together with Musser and later with Meng Le in 1992, Stepanov experimented with many architectural and algorithmic formulations in C and C++ and contributed to Stepanov's HP project.

In November 1993, Stepanov presented his work to the ANSI/ISO committee for C++ standardization. The committee's response was so overwhelmingly favorable that it led to a formal proposal for the following committee meeting in March 1993.

At that time, Hewlett-Packard decided to continue working on Stepanov's implementation of the STL and considerably improve it. In August 1994, the decision was made to make the implementation freely available. Nowadays, HP implementation is the basis of many implementations used by compilers and other libraries.

2.4.3. C++ Committee

In 1990-91 the ISO C++ committee was formed and was called WG21 (working group 21), officially ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) /

WG21 [fSb]. The committee consists of accredited experts from member nations of ISO/IEC JTC1/SC22 who are interested in C++ work.

Every year the WG21 organizes three full week-long face-to-face meetings. Traditionally, two meetings are held inside the continental United States. The third one is often in Europe but periodically in Canada, Hawaii, the Caribbean, Japan, or Australia.

The meetings are always six days long (Mon-Sat) and start with everyone in the same room for a planning session [fSa]. With everyone together, the work is organized for the rest of the week. At the end of Saturday, the committee will habe decided what new features will be included in the approval polls. There is technical discussion in smaller subgroups for the rest of the week.

2.4.4. ISO IEC 14882

Inside the ISO/IEC 14882, the requirements for implementation of the C++ programming language are specified. Since 1998 six editions have been published. The first is called ISO/IEC 14882:1998. The first requirement of every edition is that they implement the language, which also defines C++ itself. Inside the ISO/IEC 14882, besides other requirements, the STL and the synopsis of the individual STL header files are also defined.

The current ISO ISO/IEC 14882[1] from 2020 defines C++ as a general-purpose programming language based on the C programming language described in ISO/IEC 9899:2018 Programming languages — C (from now on, referred to as the C standard). C++ can be described as a superset of C and extends it by including additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, storage management operators, and additional library facilities.

There already are many implementations of STLs which are following the ISO standard like:

- GNU C++ Standard Library (libstdc++)
- LLVM C++ Standard Library (libc++)
- NVIDIA C++ Standard Library (libcu++)
- Microsoft C++ Standard Library (MSVC STL)

[1] https://www.iso.org/standard/79358.html

Part II.

Thesis Development

3. The requirements to make LAMMPS compile

As already mentioned in the Chapter 2 the code base of LAMMPS is written in C++, more specifically the C++-11 standard, plus LAMMPS uses the MPI library, which is already defined inside OctoPOS. In addition, to compile, we need either CMake or `make`. The advantages and disadvantages of the two combined with LAMMPS are described in the following section.

3.1. LAMMPS build

To build LAMMPS, we have two options, either use CMake or `make`.

3.1.1. CMake

CMake is entirely open-source and is used to build, test, and package software on many different platforms [Kit]. CMake is used to control the software compilation process using simple platform and compiler independent configuration files and generate native makefiles and workspaces that can be used in the compiler environment of anyone's choice.

LAMMPS lists the following advantages to use CMake in its documentation [LAM21]:

- Using CMake is beneficial for people with limited experience in compiling software.
- CMake makes it also easy to extend the existing CMake configurations
- CMake will adapt the LAMMPS default builds according to the detected available hardware, tools, features, and libraries.
- The settings customization can be done in text mode or with a graphical user interface, both are supported by CMake.
- With a single operation, all enabled components can be compiled.
- For all files and configuration options, we have an automated dependency tracking system.

3.1.2. make

This thesis did not need any of the previously mentioned features, so we modified and extended an existing makefile and used `make` directly, which is the traditional way of compiling LAMMPS. Other OctoPOS applications use pre-defined makefiles to compile, thus it made sense to make also use of it in combination with LAMMPS. The makefile for serial LAMMPS execution was used as a base, and the OctoPOS makefile gets included. LAMMPS and OctoPOS use different variables for the same thing, and all import variables are listed

makefile variable names			description
OctoPOS	-	LAMMPS	
CXX	-	CC	compiler for C++-files
CXXFLAGS	-	CCFLAGS	flags for C++-files
	-	SHFLAGS	shared flags for C++-files
	-	DEPFLAGS	dependency flags
	-	LINK	compiler for linked files
LDFLAGS	-	LINKFLAGS	flags for linked files
	-	LIB	name of MPI library
	-	SIZE	
AR	-	ARCHIVE	command for archives
	-	ARFLAGS	archive flags
	-	SHILIBFLAGS	shared library flags
CPP_SRC	-	SRC	C++-files
C_SRC	-		C-files
CXX_SRC	-		Cxx-files
OBJECTS	-	OBJ	object files

Table 3.1.: Makefile variable description

in the Table 3.1. We modified the LAMMPS makefile but kept the LAMMPS naming convention and extended the LAMMPS variables by the OctoPOS equivalent ones. For the variables which define compilers, there the OctoPOS variables are used. By extending or overwriting the LAMMPS variables, all other makefile commands and configurations can be used and do not have to be modified. This also includes the original LAMMPS commands to compile everything together.

3.2. LAMMPS dependencies

LAMMPS uses the standard template library extensively and therefore depends on the header files, which are listed in Appendix A. There are different ways in which one could get a list of all needed dependencies. There are static code analyzers and special compiler flags.

Static code analyzers are mainly used to examine code, even before it gets compiled and executed. For example, with static analysis, weaknesses in source code, leading to vulnerabilities can be identified. That also could be done by manual code reviews of merge requests, but automated tools, like the open-source static code analyzer Cppcheck[1], are much more effective. Nonetheless, they usually cannot replace the manual code review process. The feature we are interested in is the static dependency analysis. It is mainly used to find dependency cycles and display the dependency in a graph. Furthermore, with it, we have a list of all dependencies.

We only need to know the std header files and the c standard header files, which the LAMMPS serial implementation uses. Since LAMMPS uses g++ to compile C++ and link files, we can simply use the `-H` flag of `g++`[2]. With the `-H` flag, the compiler prints the

[1] `https://github.com/danmar/cppcheck/`
[2] `https://linux.die.net/man/1/g++`

name of each header file used and other regular activities. It also indicates the depth in the `#include` stack by the indentation. Because of that, we do not need any additional tools or software to assist us.

For example we have the simple hello world code in Listing 3.1 written in C++:

```
1 #include <iostream>
2
3 int main(){
4     std::cout << "Hello World!" << std::endl;
5     return 0;
6 }
```

Listing 3.1: C++ hello world example

And in the Listing 3.2 we have the first few lines of the compiler output form `g++ -H helloWorld.cpp`, when compiled on a Ubuntu 20.04.

```
. /usr/include/c++/9/iostream
.. /usr/include/x86_64-linux-gnu/c++/9/bits/c++config.h
... /usr/include/x86_64-linux-gnu/c++/9/bits/os_defines.h
.... /usr/include/features.h15
..... /usr/include/x86_64-linux-gnu/sys/cdefs.h
...... /usr/include/x86_64-linux-gnu/bits/wordsize.h
...... /usr/include/x86_64-linux-gnu/bits/long-double.h
..... /usr/include/x86_64-linux-gnu/gnu/stubs.h
...... /usr/include/x86_64-linux-gnu/gnu/stubs-64.h
... /usr/include/x86_64-linux-gnu/c++/9/bits/cpu_defines.h
.. /usr/include/c++/9/ostream
... /usr/include/c++/9/ios


...
```

Listing 3.2: C++ hello world compiler output

In the very first line of the compiler output in the Listing 3.2, we find out that the <iostream> header file can be found inside the /usr/include/c++/9/ folder, as are all other STL header files. In addition, all files located directly in /usr/include are the standard C header files, which might be used by the STL header files or are used by LAMMPS directly.

For the thesis, we added the `-H` flag temporally to the original serial makefile of LAMMPS. This way, we get all headers that are used and all headers that the headers use, and so on. After we got the dependency list in the format of the output of the Listing 3.2, we removed the dots and the spaces at the beginning of the file paths, sorted them by name, and deleted the duplicates. In the end, we have a complete list of all LAMMPS dependencies.

4. Implementation of the libraries

LAMMPS depends on a lot of different header files from the STL and as well as C standard header files. All needed files are listed in the Appendix A. LAMMPS depends on 43 different STL header files and 21 standard C header files. However, it is worth mentioning that most C files are only indirectly accessed over a wrapper in the STL files.

In addition, from this point onward, we will call this special implementation of the STL for OctoPOS **liboctoc++**. It is a combination of library, OctoPOS, and C++, and is similar to other STL implementation naming conventions, like the GNU libstdc++.

4.1. Implementation process

As already mentioned in the Chapter 2 the STL is described in the ISO/IEC 14882. Because LAMMPS is based on C++11, we follow the ISO/IEC 14882:2011 standard requirements. Inside the standard, the synopsis of the header files is written down, as well as how the functions and classes should behave, but not directly how they should be implemented. The synopsis defines all functions, classes, and interfaces that can be accessed in the std namespace.

4.2. Implementation examples

For this thesis, a lot of different header files were implemented, which are all listed in the Appendix A. Out of all of them, we selected the `std::stable_partition` function from the <*algorithm*> header file and the `std::map` class to be our examples, because both of them are non-trivial to implement, and therefore it took more time than other parts to develop them. Plus the `std::set` is based on the same tree data-structure as the **std::map** class, and therefore the benchmarks look similar.

4.2.1. stable partition

One of the many algorithm functions is the `std::stable_partition`, located in the <algorithm> header file. The `std::stable_partition` reorders the elements in the range $[first, last)$ in such a way that all elements for which the predicate `pred` returns true precede the elements for which the predicate `pred` returns false [cpp]. The relative order of the elements is preserved. The `std::stable_partition` function from the <algorithm> header file is described in the Partition subsection inside the ISO/IEC 14882. The synopsis of the function is defined as follows:

```
1 template<class BidirIter, class Predicate>
2 BidirIter stable_partition(BidirIter first, BidirIter last, Predicate pred);
```

Listing 4.1: `std::stable_partition` synopsis

The standard also defines the effects, returns, and complexity of this function with [2211]:

Effects: Places all the elements in the range $[first, last)$ that satisfy pred before all the elements that do not satisfy it.

Returns: An iterator i such that for any iterator j in the range $[first, i)$, $pred(*j)! = false$, and for any iterator k in the range $[i, last)$, $pred(*k) == false$. The relative order of the elements in both groups is preserved.

Complexity: At most $(last - first) * log(last - first)$ swaps, but only a linear number of swaps if there is enough extra memory. Exactly $(last - first)$ applications of the predicate.

Following the standard requirements, there are two possibilities to implement it: either in-place or with additional memory space.

When the additional space is available, the algorithm is quite simple. This is just a simplified explanation, not how it is implemented in libstdc++. Let us define N as the number of elements that equals $(last - first)$. Start with allocating N additional temporary elements. Then count the elements where the predicate returns true. This way, we know the offset of the elements which are returning false. Now we swap from the input range to the temporary allocated memory. Because we know the offset by this point, we also know to which the elements belongs. In the end, we move all elements back to the input range in the order in which they are in the temporary memory section. The last thing is to free the allocated temporary memory and return the iterator to the first element of the second group, based on the offset.

Alternatively, one could use the in-place method. For this, we modify the in-place merge-sort algorithm because merge-sort is stable. Unlike the typical merge-sort, we do not compare two elements to see which one is smaller than the other. Rather, we check if both elements are in the same group. If not, we swap them. Everything else is the same as the typical in-place merge-sort algorithm.

```
1  namespace std {
2  namespace par {
3  template <class BidirectionalIter, class Pred>
4  void inplaceMergePartition(BidirectionalIter first, BidirectionalIter middle,
5                             BidirectionalIter last, Pred pred) {
6    if (first == last || middle == last || first + 1 == last)
7      return;
8
9    if (!pred(*middle))
10     return;
11
12   BidirectionalIter pointA = std::partition_point(first, middle, pred);
13   BidirectionalIter pointB = std::partition_point(middle, last, pred);
14
15   std::rotate(pointA, middle, pointB);
16 }
17
18 template <class BidirectionalIter, class Pred>
19 void mergeSortPartition(BidirectionalIter first, BidirectionalIter last,
20                         Pred pred) {
21   if (first == last || first + 1 == last)
```

```
22        return;
23
24      BidirectionalIter mid = first + (last - first) / 2;
25      mergeSortPartition<BidirectionalIter>(first, mid, pred);
26      mergeSortPartition<BidirectionalIter>(mid, last, pred);
27      inplaceMergePartition<BidirectionalIter>(first, mid, last, pred);
28    }
29    } // namespace par
30
31    template <class BidirectionalIter, class Pred>
32    BidirectionalIter stable_partition(BidirectionalIter first,
33                                        BidirectionalIter last, Pred pred) {
34      first = std::find_if_not(first, last, pred);
35
36      if (first == last)
37        return first;
38
39      par::mergeSortPartition(first, last, pred);
40
41      return std::partition_point(first, last, pred);
42    }
43    } // namespace std
```

Listing 4.2: `std::stable_partition` implementation of liboctoc++

The GNU C++ standard library implementation tries to use the first algorithm first, and if the allocation of memory fails, it falls back to the in-place algorithm. The in-place algorithm is in the best-case scenario as fast as the alternative. Nevertheless, depending on the predicate function, it can be slower.

The liboctoc++ implementation can be seen in the Listing 4.2. The `std::stable_partition` function starts out with searching for the first element, for which the pred function returns false. If the input is empty at this point, the function ends here. Otherwise, a custom merge-sort gets called, and at the end, the partition point of the input is returned.

4.2.2. map

The `std::map` class from the <map> header file is a handy container inside STL. The `std::map` is defined as a kind of associative Container that supports unique keys (contains at most one of each key) and provides fast retrieving values of another type T based on the keys [2211]. The map class supports bidirectional iterators.

According to the standard, `std::map` has to meet the requirements of Containers, AllocatorAwareContainers, AssociativeContainers, and ReversibleContainer. An Object that is used to store other objects and take care of the memory management is called a Container. When a Container also has an instance of an Allocator, and when the Container uses that instance in all its member functions to allocate and deallocate memory and construct and destroy an object in that memory, then such Container is an AllocatorAwareContainer. AllocatorAwareContainer can handle objects like container elements and nodes (or, for unordered containers, bucket arrays). An AssociativeContainer orders its elements based on its keys to have a fast lookup capability of the inserted objects. A Container that has either a bidirectional or random-access iterators is also called ReversibleContainer because it can be iterated in both forward and backward order.

The synopsis of the **std::map** class is defined as follows in Listing 4.3, with the third and fourth template variables predefined with the std::less function to compare the keys and the allocator for the key-value pair. The complete **std::map** class and public member functions' synopsis can be found at the cppreference.com website[1].

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

Listing 4.3: **std::map** class synopsis

The std::map class has the following sections, types, constructor/copy/destructor, iterators, capacity, element access, modifiers, observers, map operations, and non-member functions like lexicographically comparing the values and specialization of the **std::swap** algorithm.

All public **std::map** member types are listed in the Table 4.1. On the base of them, we have the `key_type` and `mapped_type` for the key and value variables, respectively. The `value_type` combines these two into one pair type. The `size_type` is used to count the elements in a map, and the `difference_type` is for the distance between two map elements. The iterators have to be implemented to move in both directions, i.e., incremented and decremented.

Member type	**Definition**
`key_type`	`Key`
`mapped_type`	`T`
`value_type`	`std::pair<const key_type, mapped_type>`
`size_type`	Unsigned integer type (usually `std::size_t`)
`difference_type`	Signed integer type (usually `std::ptrdiff_t`)
`key_compare`	`Compare`
`allocator_type`	`Allocator`
`reference`	`value_type&`
`const_reference`	`const value_type&`
`pointer`	`std::allocator_traits<Allocator>::pointer`
`const_pointer`	`std::allocator_traits<Allocator>::const_pointer`
`interator`	LegacyBidirectionalIterator to `value_type`
`const_iterator`	LegacyBidirectionalIterator to `const value_type`
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

Table 4.1.: map member types with its definition and description

For the member functions, the ISO/IEC 14882 defines their effects and their complexity. For example, for the default constructor, the effect should be to construct an empty map using the specified comparison object and allocator, and its complexity should be constant. The standard directly defines the implementation for others, like the element access member

[1] https://en.cppreference.com/w/cpp/header/map

function with the square bracket operator. For the `T& operator[](const key_type& x);` function, the standard bases it on the insert function, which itself has a complexity of $O(log(n))$:

```
(*((insert(make_pair(x, T()))).first)).second
```

Listing 4.4: map access `operator[]` function return

Maps are usually implemented as red-black trees. The self-balancing binary search tree gets its name by storing an extra bit that represents "color", which is either "red" or "black". After each modification, the tree gets re-balanced by following predefined rules to restore the coloring properties. Through the re-balancing searching, inserting and deleting elements are guaranteed to have a runtime of $O(log(n))$, even if the re-balancing is not perfect.

```
namespace std {
template <class key_type, class mapped_type> class RedBlackTreeMap {
public:
    ...
    NodePtr insert(const std::pair<key_type, mapped_type> &p);
    ...
}

template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T>>>
class map_base {
public:
    ...
  iterator insertValue(const value_type &value) {
    return TreeIteratorMap<key_type, mapped_type, value_type>(
        &tree, tree.insert(value));
  }
    ...
private:
    RedBlackTreeMap<key_type, mapped_type> tree;
}

template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T>>>
class map : private map_base<Key, T>{
public:
    ...
  // element access
  mapped_type &operator[](const key_type &x) {
    return (*((insert(make_pair(x , mapped_type()))).first)).second;
  }
    ...
  pair<iterator, bool> insert(const value_type &x) { return insert_helper(x);
      }
    ...
private:
  pair<iterator, bool> insert_helper(const value_type &x) {
    auto node = find(x.first);
    if (node != end())
      return std::pair<iterator, bool>(node, false);
    iterator p = map_base<Key, T>::insertValue(x);
    return std::pair<iterator, bool>(p, p != end());
```

```
42      }
43  }
44  }
```

Listing 4.5: `std::stable_partition` implementation of liboctoc++

The Listing 4.5 shows the code which is used to insert elements into an `std::map` instance. It also shows how the class depends on other classes. The `std::map` class is derived from the `map_base`, because there is the `std::multimap` class as well. The `std::multimap` shares a lot of logic with the normal `std::map`, and to have a DRY (Don't Repeat Yourself) code, there has to be a base class. The base class mainly handles the access to the red-black tree data structure. Furthermore, to insert a new element over the `operator[]`, we try to insert a new element with the default value of the `mapped_type` with the key x. If the map has no entry under the key, a new element will be inserted. Otherwise, the reference of the already existing element will be returned, ready to be read, or modified.

4.3. Library dependencies

As mentioned before, the standard template library has some C wrapper header files. In the Figure 4.1 all 16 of them that are needed are displayed with their dependencies. Nearly all of them only depend on the C header file for which they are the wrapper. The round nodes are the C++ STL headers, and the C header files are hexagons.

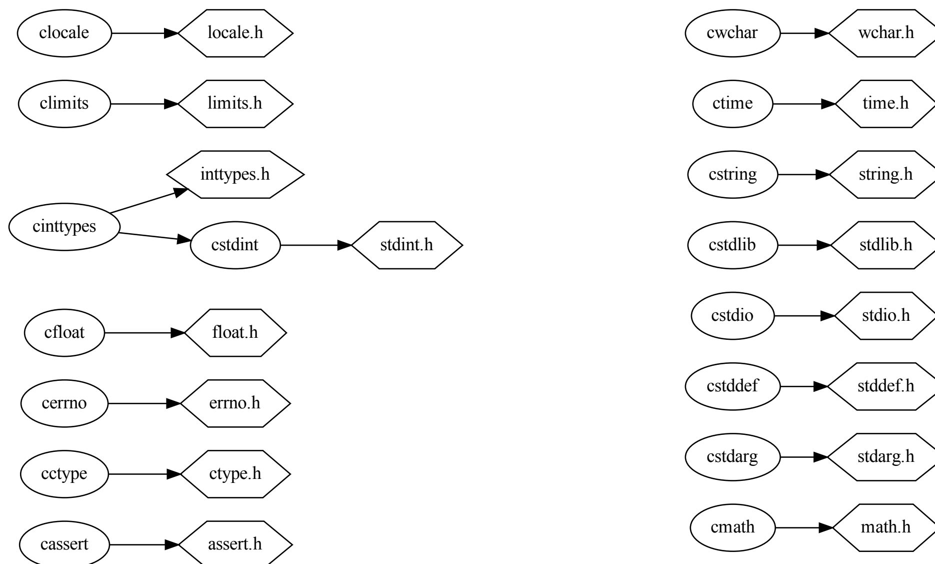


Figure 4.1.: LAMMPS C wrapper header files

The C wrapper dependencies are pretty much straightforward, unlike the rest of the

STL header files, which can be seen in the Figure 4.2. However, the graph here is highly simplified. In reality, the STL headers are very interconnected. For example, the <map> header file from the container libraries depends directly on the <memory> header, the <iterator> header file, and the `std::pair` struct from the <utility> header file. The complete dependencies graph is in the Appendix B.

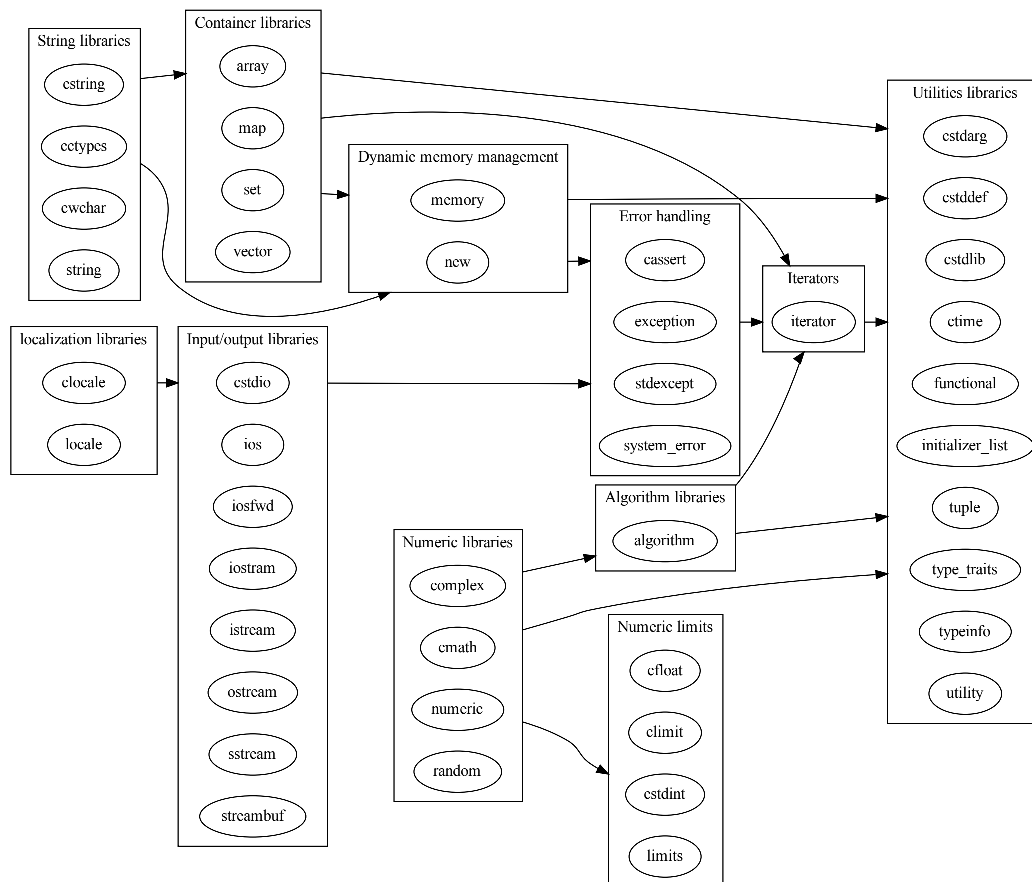


Figure 4.2.: LAMMPS STL dependencies simplified

4.4. Implementation references

All the STL are defined in the ISO/IEC 14882 standard, and the current edition 6 is sold by the independent and non-governmental international ISO organization for 198,– CHF. The 2011 version of the standard that got retracted is no longer purchasable over the ISO store, but the ASIE store still sells it for 60,–USD. Moreover, the ISO/IEC 9899 is also needed, which defines the C language itself and its header files, on which a lot of the STL components are based. Furthermore, the PDF for this standard is also sold for 198,–CHF at

the ISO store. Fortunately, there are also free alternatives, like the docs from IBM. We find the structure and defined macros of standard C header files in them.

A much better resource is the cppreference.com website. The site was created and is maintained by a group of C++ enthusiasts from around the world. It is set up like other wiki pages, and everyone can contribute to it. According to the site's about page, its goal is to provide programmers with a complete online reference for the C and C++ languages and standard libraries, i.e., a more convenient version of the C and C++ standards. The standard library descriptions are significant for this thesis.

The site not only lists all STL header files and their synopsis but also sometimes simple implementations, which might not be the most efficient ones, but are a helpful base. In addition, they list which part of each header file got added with which C++ iteration and a detailed description of every function variation. As well as add a comment if a part is deprecated or if it got replaced with something else.

All the content of the cppreference.com website is licensed under the Creative Commons Attribution-Sharealike 3.0 Unported License (CC-BY-SA). By the GNU Free Documentation License, we can share and alter the content, but only if we attribute the website and distribute the modified code with the same, similar, or compatible license.

In most cases, we did not have a ready-to-copy implementation on cppreference.com. Therefore we either implement the algorithms according to the description or, alternatively, we look at other already implemented STL libraries. A reference implementation that was used at the beginning of the implementation of liboctoc++ was the uClibc++[2], which is a subset of the C++ standard library and is especially designed for embedded systems. uClibc++ is well structured and is simple to understand, but unfortunately, its code is not entirely ISO conforming when compared with the definitions from cppreference.com. It is notable is that the `std::string` and `std::vector` implementations from uClibc++ were used and adapted to be ISO conforming and is now used inside liboctoc++. In the cases where the cppreference.com and uClibc++ could not help, there was also the STL implementation by GNU. The libstdc++ is, of course, ISO conforming and highly optimized. Although, its code is very complicated and structured with many helper functions, it is still an excellent source, especially in the moments where the cppreference.com reference implementations uses `decltype`, which is not supported yet by OctoPOS, and GNU does not use them in their implementation.

For this thesis, around 20000 lines of code were added to the liboctoc++ library. Around a third of it is just the synopsis directly copied from the cppreference.com website. When we take the <algorithm> header file, for example, the synopsis of all the algorithm functions is around 900 lines, and the implementation has around 1500 lines of code. Furthermore, to test all the newly added functionality, around 6000 lines of test code were added, which are described in the upcoming Chapter 7.

[2] `https://cxx.uclibc.org/`

5. Testing framework

5.1. Catch2 testing framework

For testing, we use the testing framework Catch2 [Org]. Catch2 is mainly a unit testing framework for C++, but it also provides basic micro-benchmarking features and simple Behavior-Driven Development (BDD) macros. BDD is a variation of test-driven development (TDD), where functions are first described as a story from the user's point of view, and then a test is created and implemented based on this.

Catch2 describes itself as a simple testing framework that feels natural when using it. The Catch2 test cases, written in regular C++ code, auto-register themselves. The tests do not have to be named with valid identifiers, and sections provide an excellent way to share set-up and tear-down code in tests.

5.1.1. Why use this testing framework?

Catch2 is not the only established C++ framework out there. The most used one is Google Test, but there also exists Boost Test, CppUnit, Cute, and many, many more.

Besides having a catchy name, the Catch2 documentation lists the following key features as remarkable for the test framework:

- It is quick and easy to get started. For example, by simply adding Catch2 as a git sub-module.
- Besides a compiler with C++14 support, there are no other external dependencies. (the older version v2.x depends on C++11)
- Every test case is a function that registers itself.
- Test-cases can be grouped into sections, each run in isolation. That way, there is no need for fixtures.
- Uses BDD-style Given-When-Then sections as well as traditional unit test cases.
- The framework uses the default assert function from STL wrapped in a macro for the test elements. At the same time, the entire expression is decomposed, and lhs and rhs values are logged.
- Tests are named using free-form strings, that means there is no strict naming rule for the tests.

Other mentionable core features are:

- Besides grouping test cases with sections, they can also be grouped by tags for quickly running ad-hoc groups of tests.

- The test result outputs are handled by modular reporter objects, which output basic textual and XML reports. Additional output formats can easily be added.
- For third-party tools, like a CI server, JUnit XML output is included as well.
- The latest version of Catch2 provides a default main() function, which a custom one can replace (e.g., integration into your own test runner GUI).
- Even if a custom main() function is used, the command line parsers will still be provided and still can be used.
- Besides the default test assertion macro, which aborts the test case if it fails, there are also other macros, which only report and count the failed test and go on executing the rest of the test case. Moreover, others are waiting for an exception itself when executing.
- A complete set of floating-point comparisons is provided by default (Catcha::Approx)
- Internal and friendly macros are isolated so name clashes can be managed
- Data generators (data-driven test support)
- Microbenchmarking support

5.1.2. Why do we use Catch2?

Lets look at the C++ ecosystem survey [1]. In the annual developer ecosystem survey, Jetbrains asked the participants, among other questions, which language they use as their primary programming language and what standard they are most comfortable with. In the case of C++, they also asked which compiler and which build system the community uses the most and which unit testing framework they use regularly. Google Test is in first place with 32%, Catch2 is in second place with 11%, and third place goes to Boost.Test with 9%. It is interesting to note that 30% of the participants said they are not writing any unit test for C++ code.

For this Thesis, we selected Catch2 over Google Test because another team already uses Catch2 for unit tests for iRTSS, so it was easy to modify it for our needs for the OctoPOS unit tests. In addition, Catch2 made the integration of the framework very straightforward.

5.1.3. Why write unit tests

There are many different reasons why we want to have unit tests. The most important ones are [Goe]:

Find Stupid Bugs Early We need unit tests to find incorrectly defined constants or wrongly implemented algorithms before it is too late. Without these tests, vulnerabilities can go unnoticed in the code and only become apparent later during integration tests or even worse in production. If we take the time and do unit tests while writing the methods or functions, such errors will be noticed in the early stages of production.

Avoid Regressions Sometimes, a bug is hidden in the production code and only appears after a patch or a system update. To prevent such a scenario running a unit test on every build helps verify that new features do not introduce bugs in the old code.

[1] `https://www.jetbrains.com/lp/devecosystem-2021/cpp/`

Get Early Feedback Unit testing will give you feedback on, among others, if you caused a new regression or whether the new code is doing what you want it to do.

Create Built-in Documentation We can always look into the test to not rely on outdated comments. As long as they run through without any errors, tests are the one thing that gets updated regularly, especially when we have continuous integration tests.

We want to write a practical test of discrete units and verify their functionality in isolation. When a failed test does not make apparent what caused the failure, the unit test is not isolated.

5.2. Testing in OctoPOS

Unfortunately, we can not use the full version of Catch2 to run tests and benchmarks with OctoPOS. Catch2 uses exceptions and catches them if a test assertion fails. The test frameworks count how many test cases were successful and failed and how many assertions were in them.

We use a subset of Catch2 that does not throw exceptions at this time point. It counts the failed assertions. The problem with this method is that it might count a failed test case multiple times if there are multiple failed asserts in one test case.

5.2.1. OctoPOS lib test concept

The idea was to test every newly implemented dependency that LAMMPS needs independent of the LAMMPS itself. LAMMPS depends on 43 C++ header and 21 C header files. Of these 39 C++ headers that needed to be implemented, for the C headers, only five were missing.

The test cases got split up into libc++ and libc tests.

In total, we have 3805 asserts in 431 tests. The libc test only has 16 asserts in 4 test cases, and the rest are libc++ tests. We only have a few libc tests because the newly implemented c headers are mainly constants and macros defined.

We call and test every newly implemented function at least once.

5.2.2. Test example implementation

Representative for other tests, the unit test implementation for `std::stable_partition` is in the Listing 5.1. The test starts with defining a container and inserting elements in a range between zero and 100. A lambda expression to check if an integer is even is defined as well. After that the `std::stable_partition` is used on the container with the `is_even` function. Directly afterwards, the `std::is_partitioned` is used to verify if the container's elements are partitioned now, but not if the algorithm was stable. Therefore we need to go through the vector and check if the relative position of the elements did not change.

```
1 #include <algorithm>
2 #include <catch2.hpp>
3 #include <vector>
4
5 TEST_CASE("test stable_partition", "[libc++-tests][algorithm][partition]")
```

```
6  {
7    std::vector<int> v = {};
8    for (int i = 0; i < 100; ++i) {
9      v.push_back(i);
10   }
11   auto is_even = [](int i) { return i % 2 == 0; };
12
13   std::stable_partition(v.begin(), v.end(), is_even);
14
15   REQUIRE(std::is_partitioned(v.begin(), v.end(), is_even));
16   for (int i = 0; i < v.size(); ++i) {
17     if (i < v.size() / 2)
18       v.at(i) = i * 2;
19     else
20       REQUIRE(v.at(i) == (i - (v.size() / 2)) * 2 + 1);
21   }
22 }
```

Listing 5.1: `std::stable_partition` unit test

Other tests, especially for components of the <algorithm>, are structured similar to the Listing 5.1. They start out by defining a container, which could be a array, vector, map or set and continues with inserting a few elements in a predefined order. Very often a custom lambda expression is defined as well. After the setup, the method which we want to test is called. And at the end of the test-case the output gets checked against the expected output.

Part III.

Results, Conclusions and Future work

6. Comparison of efficiency between OctoPOS and normal Linux

To compare the OctoPOS and a normal Linux system, the `std::stable_partition` algorithm and the `std::map` were selected. Two different input variants will be tested on different batch sizes to test their performance. First, the input elements are in order, which means the input goes from zero to N, which is the batch size. For the second test, the input is pseudo-randomized. That is achieved by selecting a bigger prime number than the batch size. We start at zero and add the prime number and use the module with the batch size. We repeat that process for the whole batch. Because we use a prime number, we get every number only once.

The liboctoc++ implementation is compared against the GNU libstdc++ with the g++ compiler set to the C++11 language standard with `-std=c++11`. Additionally, all benchmarks got compiled with the `-O3` optimization flag. This way, the compiler attempts to improve the performance at the expense of compilation time and output code size. With `-O3` we turn on all optimizations specified by `-O2`, that also includes `-O1` optimizations, as well as additional optimization flags like `-fguess-branch-probability` and many more [1].

For each performance test we execute the code multiple times, measure the time with `clock()` from the C header file <time.h>, and take the average. The smaller test sizes are executed a few thousand times and the bigger batch sizes are executed at least sixteen times. In addition we also can calculate the standard deviation. We used the following formulas, with N as the batch size and x_i as the i-th execution time.

$$\mu = \frac{1}{N} \cdot \sum_{i=1}^{N} x_i$$

$$\sigma = \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (x_i - \mu)^2}$$

6.1. stable partition

As already mentioned before, `std::stable_partition`, from the <algorithm> header file, reorders all the elements in the range [first, last), that all elements for which a binary function, called pred for the predicate, returns true, are placed before all elements for which the pred returns false. Additionally, the relative order of the elements has to be preserved.

What we can see in the Figure 6.1 and what was evident from the beginning was that the partition of the already sorted array is faster than in a more realistic scenario. However, the

[1] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

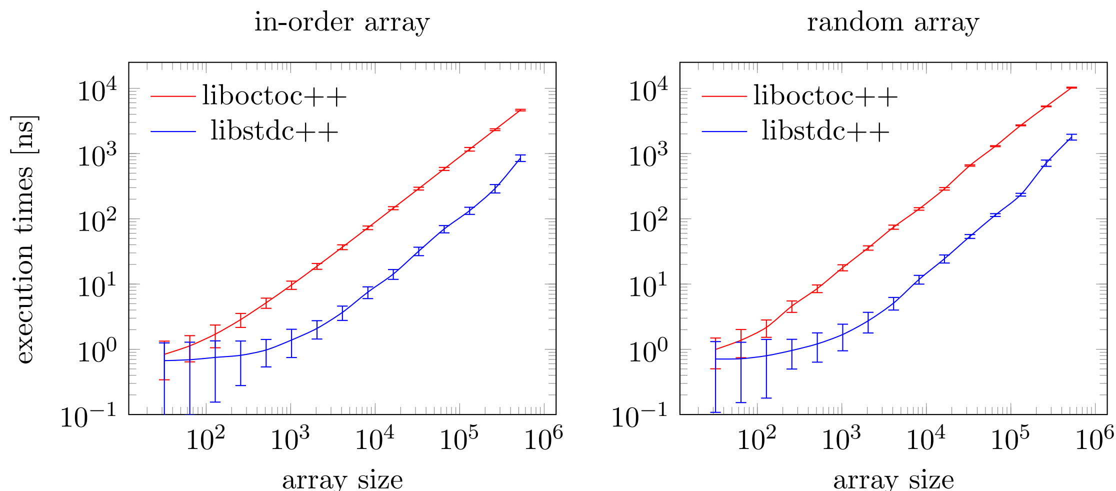


Figure 6.1.: Stable partition benchmark comparison: x86guest liboctoc++ vs C++11 libstdc++

more exciting part is that the time difference between the g++ in-order and pseudo-random benchmark is minimal, and both grow linear with the array size. This indicates that there are only a linear number of swaps and that the algorithm allocates additional memory, like it is expected by the ISO/IEC 14882.

On the other side, the OctoPOS implementation does not try to allocate any memory. This way, it always uses the in-place method. Because the method does not allocate additional memory, and in the case of the ordered array benchmark, the algorithm does not need to swap any elements. Nevertheless, even without swapping the elements, the OctoPOS implementation is multiple times slower than the reference implementation. Just going through the array and making `N*log(N)` times many comparisons takes much longer than creating a new memory block and moving the elements to the right place after finding the right offset.

6.2. map insert

For the benchmark, we insert N pairs into the map. As for the benchmark before, we added the elements once in order and the second time in randomized order. When the elements are inserted in order, we force the underlying self-balancing tree data-structure to re-balance periodically. A restructure of the tree is needed, because we only add elements at the right side of the tree, and without it, we would end up with a linked list, with the smallest element as root. In the case, when we insert the values randomly, the re-balancing does not have to be done that often. Based on that the expectation is that, the execution time of the randomly inserted elements is slightly faster, because the tree structure is better fitted for such inputs.

The GNU and the OctoPOS implementation use a red-black tree as a base for the map class. If the tree gets “too unbalanced”, the tree fixes itself. In the GNU implementation, we

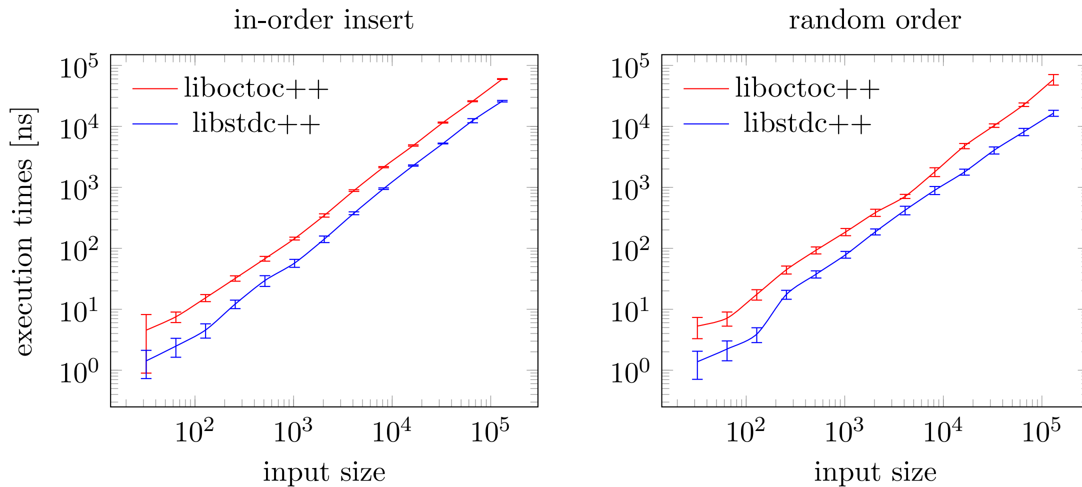


Figure 6.2.: Map insert benchmark comparison: x86guest liboctoc++ vs C++11 libstdc++

can see a point where the random inserts are no longer slower than the in-order executions. The reason is that at some point, the tree does not need to be fixed after a couple of inserts because the randomness of the input spreads the new elements up to the different leaves.

Even when, in principle, both implementations are based on a red-black tree, the GNU implementation is around three times faster, as can be seen in the Figure 6.2. The main reason is that OctoPOS uses a simple implementation of the red-black tree without any optimizations and hardly ever uses hints, which indicate where an element might be inserted next.

6.3. x86guest vs x64native

So far the stdlibc++ got compared against the liboctoc++ on x86guest. The reason is mainly for convenience. With x86guest, we can execute the benchmark like any other program on the host system. Furthermore, the results can be copied and pasted, unlike with x64native, where we load the tests via a PXE boot server and therefore have to tip off the test results. Alternatively, the output data can be extracted form a photo using freely available Optical Character Recognition (OCR).

The expectation was that x86guest was slower than the x64native variant because x64native directly runs on the hardware. As all comparisons show, in Figure 6.3 and Figure 6.4. The x64native benchmarks start faster with tiny input sizes, which is explainable by not having the overhead from a host system and not having additional code to support that.

With the `std::stable_partition` benchmark the x86guest is already faster on an input size of 256 elements. And when inserting elements into an `std::map` class, x86guest gets faster after 2048 inserted elements.

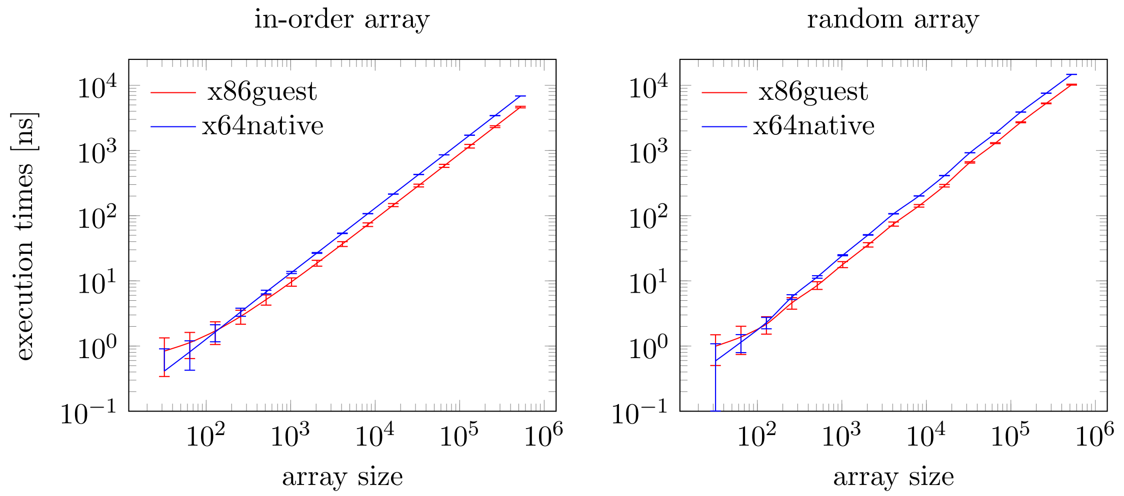


Figure 6.3.: Stable partition benchmark comparison: x86guest vs x64native

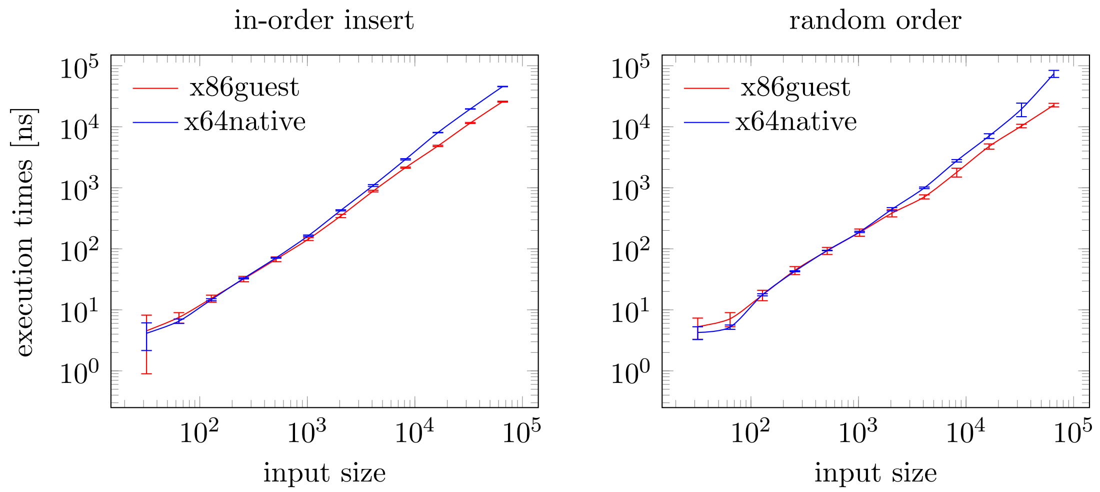


Figure 6.4.: Map insert benchmark comparison: x86guest vs x64native

7. Testing

To test the source code, a modified version of Catch2 is used. The modification was needed because the testing framework is based on C++14, and so far, OctoPOS only supports the C++11 standard. Plus, OctoPOS does not support throwing and catching exceptions, which the testing framework uses to count the failed test cases. Catch2 takes the test expression and asserts them. If the assertion fails, an exception gets thrown, and the test case gets counted and logged as failed.

For this thesis a test-driven development was used. First, a test was created for the feature, which was implemented next. The tests were checked by running them against the GNU stdlibc++ implementation. In total, we have 431 test cases, which have 3807 assertions. These all are distributed into 49 test files, from which three are used to test standard C functions. The directory tree is displayed in the Appendix C. In one of the C files, `strtok` from <string.h> is tested and the second one checks `strcasecmp`/`strncasecmp` from <strings.h>. These three functions were validated, because they were missing at the time of the implementation of liboctoc++. In the `all-c.cc` and `all-c++.cc` all C and C++ from the libc and libc++ folders are imported respectively, to check for redefinitions of functions and classes. Every newly implemented header file of the liboctoc++ has its own test file, with the corresponding name. In addition the unit-tests of <algorithm> were grouped into ten category and every one of them is in a separate file. And <map> was split into two files for testing `std::map` and `std::multimap`. <set> was similar, but for `std::set` and `std::multiset`.

All test cases are equipped with tags, which allow the grouping of the related test together. In the full version of Catch2, tags are used to select tests (for running or just for listings) and even by an expression that combines several tags. We do not have that functionality in the modified version. Currently, it is only possible to run all test cases. The test framework entirely ignores the tags, but they might be helpful in the case when the framework is upgraded to the full version.

7.1. Tests build setup

The tests are set up like other OctoPOS apps and can be built over the `make` command. Like with other apps, we can change the target architecture and architecture-specific variant with the build parameters `ARCH` and `VARIANT` respectively.

There also exists a new Docker image, structured as a multi-stage build. The first of two stages is for iRTSS and starts by installing all necessary dependencies for iRTSS, and afterward, iRTSS itself gets built. The build target can be influenced by the build argument `IRTSS_PLATFORM`, which defines which platform variant configuration should be used. By default it will be built for the x86guest architecture and the generic-debug variant. In the second stage, the dependencies are installed again, this time only the ones needed for the OctoPOS. Inside the docker, the x86guest variant can be executed as expected. If the

x64native architecture is needed to be executed, QEMU is also installed in the docker image. Although QEMU needs to have read and write access to the KVM device of the host system. A user can start up a container with privileged rights, to lift all limitation enforcements and give the running container all rights to access the KVM device. However, a better solution is to give the container only access to devices it needs, i.e., only the `/dev/kvm`.

7.2. Test execution

In the following example output in the Listing 7.1, the tests get compiled and run inside a docker container. With `make` the tests are getting compiled for the default architecture, which in this case is x86guest and the generic-debug variant. Furthermore, we can run through the tests after it is compiled simply by executing the test_runner file.

The test starts OctoPOS and waits for the tile synchronization, but only executes the test on the tile zero and does not do any tests in parallel. Moreover, there is no checker that tests the liboctoc++ against race conditions or any other problems associated with executing code in parallel. Which is not usually tested by unit tests anyway.

```
user@docker:/iHPC/octopos/tests$ make
user@docker:/iHPC/octopos/tests$ ./test_runner

125 MiB of TLM free, 15 MiB of SHM free
iRTSS - release.x86guest.generic-debug -
   v2019-03-960-g8633ce524-dirty - tile type: compute
iRTSS built at 2022-03-25 12:25:31 (Build-ID: 2793148034)
iRTSS committed at 2022-03-24 (authored: 2022-03-24)
iRTSS configuration:
Scheduler: Work-stealing scheduler with
   lib::adt::WorkStealingQueue with capacity 64
Tiles: 16 - Cores: 15
Tile ID 0 of 15 ready for tile sync
Waiting for tile sync:  [ Tile sync complete ]
OctoPOS startup completed.
This is a catch-mini host application

===============================================================
All tests passed (3807 assertions in 431 test cases)
```

Listing 7.1: unit test result of x86guest

7.3. Shortcomings

Because of the time limitations of the thesis, every implemented method, on average, only has one test case with multiple assertions. Often edge cases were not tested, like empty, random, or massive inputs. So far, the tests mainly verify that liboctoc++ works with the primitive built-in data types from C++. It does not mean it would not work with user-defined data types, but there are hardly any tests with user-defined structs and classes.

8. Conclusion

The overall goal of this thesis was to implement all missing dependencies of LAMMPS. More specifically, adding a custom C++11 STL implementation to OctoPOS. The thesis started by analyzing the build process of LAMMPS and getting a complete list of all dependencies. By comparing the LAMMPS dependencies list against the standard header files that OctoPOS had already implemented, we determined which header files are still missing from OctoPOS.

Test-driven development was chosen as the software development process, and a unit-testing framework was integrated into the OctoPOS project. This way, we enforced that every newly developed component of liboctoc++ has at least one unit test. The unit-testing framework Catch2 was selected for its ease of use and for being easy to modify and integrate.

The cppreference.com website provided the missing STL definitions. The interfaces of the new components of the liboctoc++ are ISO-conforming. Due to time constraints, not everything of liboctoc++ got implemented according to the standard. Through the unit tests, we tested if the functionality of the liboctoc++ is correct, but it does not check for the time and space complexity of the components.

During the implementation of liboctoc++, the functionality was the focus, and the performance suffered. That is why there is a significant performance difference between liboctoc++ and GNU's libstdc++. In hindsight, it is evident that a self-developed library can not compete against a library in development for decades and has a massive community behind it.

A point that could not be answered in this thesis is why the execution on bare-metal with the x64native target is slower than on the guest system with additional overhead to execute. The expectation was that the program should perform better without the guest target overhead. However, as the comparisons have shown unless the input size is tiny, the guest target is slightly faster.

Furthermore, it was worth implementing a separate STL implementation for the last point. It depends on the further goals of the STL implementation. When we need complete control over the implementation and want to guarantee compatibility, we need to implement and test every component. However, it needs more work and even more testing to achieve that.

The goal of the thesis was to implement a custom version of the STL. Another possibility would have been to use an already existing implementation and make it work with OctoPOS. It all depends on the further goals of liboctoc++. If complete compatibility and total control over the implementation is a high priority, then liboctoc++ needs to be extended. However, if the performance is of a higher significance, it would make sense to invest time to integrate the STL implementation from GNU, stdlibc++, for example.

9. Future work

The current implementation of liboctoc++ still has room for improvements, and it is not feature-complete yet.

9.1. Performance improvements

As seen in the previous Chapter 6, some implementations are much better optimized and often switch between different algorithms depending on the input. For example for `std::sort`, stdlibc++ uses IntroSort. IntroSort is a hybrid of QuickSort, HeapSort, and InsertionSort. QuickSort is used by default, and if it takes longer than $N * log(N)$ because of unfair partitioning, it will be switched to HeapSort. Furthermore, if the input size is tiny, InsertionSort will be used. In liboctoc++ MergeSort is used, which has a worst-case run-time of $O(N * log(N))$. The main reason why MergeSort was selected is that the algorithm to merge data is also part of the STL standard. So by using the `std::merge` function, it was easy to implement MergeSort. In addition, if the STL defines an algorithm, it often also defines an in-place variant of it. Sometimes the in-place algorithm is only used as a fallback, like in `std::stable_partition` function, and other times it is separately callable, like `std::merge` and `std::inplace_merge`. In liboctoc++, to save implementation and testing time, in most cases only the in-place methods were implemented. And in the case of the two merge functions, the `std::merge` method just calls the `std::inplace_merge` function.

9.2. Library extensions

Other parts still needed to be extended, like the `std::map` container class. At this point, the map implementation meets the requirements for being a Container, an AssociativeContainer, and a ReversibleContainer. However, according to the standard, it also needs to be AllocatorAwareContainer, which means the current implementation does not use the `std::allocator` from the <memory> header file to manage the inserted elements' data.

Some other headers files are not implemented at all at this point. For example, from the container category, <deque>, <list>, <queue>, <stack>, <unordered_map> and <unordered_set> are missing from liboctoc++.

9.3. Exception handling

Another big part that is missing is exception handling. To work around that the `-fno-exceptions` flag got added, for the tests and LAMMPS compilation. The basic exception classes are implemented in the liboctoc++, like the `std::exception` class and the derived from it classes in the <stdexcept> header file, like the `std::length_error` and the `std::overflow_error`.

Nevertheless, the underlying compiler functions [1] for the exception handling are missing. For example the `__cxa_allocate_exception` library function, to allocate space to store the exception object and the exception header `__cxa_exception`. Alternatively, or as an extension, for the `-fno-exceptions` flag it is also possible to overwrite the exception allocation and raising function, and overwriting `try` and `catch` keywords. When the exception library functions are called, abort will be called, and the program ends without any feedback. The keyword `try` will be simply replaced by an `if` statement that is always true. And `catch` gets transformed into an `else`, which will never be reached.

```
1 void __cxa_allocate_exception() { abort(); }
2 void __cxa_throw() { abort(); }
3
4 #undef try
5 #undef catch
6 #define try if (true)
7 #define catch if (false)
```

Listing 9.1: example for overwriting core exception handling functions

9.4. Using libstdc++

Instead of implementing the STL from ground-up, an alternative would be to use stdlibc++ from GNU. The benefit would be that the implemented library would be standard conforming and feature-complete. However, libstdc++ expects that all standard C header files are available and that they are correctly implemented. We also have to configure what features and version we want to have activated, plus having in mind what features the hardware supports. The whole list of prerequirements is listed in the install documentation on `gnu.gcc.org` [2]. Also, exception handling is a big part of the stdlibc++, so the exception standard library functions have to be implemented never the less. It is theoretically possible to switch between the different C++ STL versions after all requirements are met. As mentioned multiple times in this thesis, LAMMPS depends on C++11, and the newest version of the testing framework we used is based on C++14.

[1] `https://maskray.me/blog/2020-12-12-c++-exception-handling-abi`
[2] `https://gcc.gnu.org/install/prerequisites.html`

Part IV.

Appendix

A. LAMMPS dependencie list

LAMMPS depends directly and indirectly on the following C++11 STL header files:

- ⋆ <algorithm>
- ⋆ <array>
- ⋆ <cctype>
- ⋆ <cfloat>
- ⋆ <cinttypes>
- ⋆ <clocale>
- ⋆ <cmath>
- ⋆ <complex>
- ⋆ <cstdarg>
- ⋆ <cstddef>
- ⋆ <cstdint>
- ⋆ <cstdio>
- ⋆ <cstring>
- ⋆ <ctime>
- ⋆ <cwchar>
- ⋆ <excpetion>
- ⋆ <functional>
- ⋆ <initializer_list>
- ⋆ <ios>
- ⋆ <iosfwd>
- ⋆ <iostream>
- ⋆ <istream>
- ⋆ <iterator>
- ⋆ <limits>
- ⋆ <locale>
- ⋆ <map>
- ⋆ <memory>
- • <new>
- ⋆ <numeric>
- ⋆ <ostream>
- ⋆ <random>
- ⋆ <set>
- ⋆ <sstream>
- ⋆ <stdexcept>
- ⋆ <stdlib.h>
- ⋆ <streambuf>
- ⋆ <string>
- ⋆ <system_error>
- ⋆ <tuple>
- ⋆ <type_traits>
- ⋆ <typeinfo>
- ⋆ <utility>
- ⋆ <vector>

and it also depended on the following standard C headers:

- • <alloca.h>
- • <ctype.h>
- ⋆ <dirent.h>
- • <endian.h>
- • <errno.h>
- ⋆ <fcntl.h>
- • <inttypes.h>
- • <limits.h>
- • <locale.h>
- • <malloc.h>
- • <math.h>
- • <pthread.h>
- ⋆ <sched.h>
- • <stdint.h>
- • <stdio.h>
- • <stdlib.h>
- • <string.h>
- ⋆ <strings.h>
- ⋆ <time.h>
- ⋆ <unistd.h>
- ⋆ <wchar.h>
- ⋆ <wctype.h>

All header files which have an "⋆" as a bullet point were implemented for this thesis. Prior to this thesis, from the C++ header files, only the <new> header file was already implemented Additionally most standard C header files were already implemented as well.

B. Complete libc++ dependency tree

C. OctoPOS tests directory tree

```
octopos
├── apps
│   └── ...
├── libs
│   └── ...
├── releases
│   └── ...
├── tests
│   ├── catch-mini
│   ├── testcases
│   │   ├── libc
│   │   │   ├── all-c.cc
│   │   │   ├── string.h.cc
│   │   │   └── strings.h.cc
│   │   ├── libc++
│   │   │   ├── algorithm
│   │   │   │   ├── binarySearchings.cc
│   │   │   │   ├── ...
│   │   │   │   └── sortings.cc
│   │   │   ├── map
│   │   │   │   ├── map.cc
│   │   │   │   └── multimap.cc
│   │   │   ├── set
│   │   │   │   ├── set.cc
│   │   │   │   └── multiset.cc
│   │   │   ├── all-c++.cc
│   │   │   ├── array.cc
│   │   │   ├── ...
│   │   │   └── vector.cc
│   │   └── test_main.cc
│   ├── Makefile
│   └── README.md
└── tools
    └── ...
```

List of Figures

2.1. State diagram of an invasive program [OSK+11] 5

4.1. LAMMPS C wrapper header files 21
4.2. LAMMPS STL dependencies simplified 22

6.1. Stable partition benchmark comparison: x86guest liboctoc++ vs C++11 libstdc++ 30
6.2. Map insert benchmark comparison: x86guest liboctoc++ vs C++11 libstdc++ 31
6.3. Stable partition benchmark comparison: x86guest vs x64native 32
6.4. Map insert benchmark comparison: x86guest vs x64native 32

Listings

3.1. C++ hello world example 15
3.2. C++ hello world compiler output 15

4.1. `std::stable_partition` synopsis 16
4.2. `std::stable_partition` implementation of liboctoc++ 17
4.3. `std::map` class synopsis 19
4.4. map access `operator[]` function return 20
4.5. `std::stable_partition` implementation of liboctoc++ 20

5.1. `std::stable_partition` unit test 26

7.1. unit test result of x86guest 34

9.1. example for overwriting core exception handling functions 37

List of Tables

3.1. Makefile variable description 14

4.1. map member types with its definition and description 19

Bibliography

[2211] ISO/IEC JTC 1/SC 22. Programming languages — c++. Standard, International Organization for Standardization, Geneva, CH, October 2011.

[Bio] Matthew Bio. A Brief History of the Multi-Core Desktop CPU. https://www.techspot.com/article/2363-multi-core-cpu/. Accessed: 2022-04-17.

[cpp] cppreference.com. C++ reference. https://en.cppreference.com. Accessed: 2022-04-15.

[fSa] International Organisation for Standardisation. Meetings and Participation: Standard C++. https://isocpp.org/std/meetings-and-participation. Accessed: 2022-03-12.

[fSb] International Organisation for Standardisation. The Committee: Standard C++. https://isocpp.org/std/the-committee. Accessed: 2022-03-12.

[Goe] Eric Goebelbecker. Unit Testing With C++: The How and the Why. https://www.typemock.com/unit-testing-with-c/. Accessed: 2022-04-16.

[Kit] Kitware. CMake. https://cmake.org/. Accessed: 2022-03-15.

[LAM21] LAMMPS. Documentation (27 oct 2021 version) - introduction. https://docs.lammps.org/Intro.html, 2021. Accessed: 2021-11-27.

[MBZ+15] Manuel Mohr, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel, and Daniel Lohmann. Cutting out the middleman: Os-level support for x10 activities. In *Proceedings of the ACM SIGPLAN Workshop on X10*, X10 2015, page 13–18, New York, NY, USA, 2015. Association for Computing Machinery.

[Mic] Microsoft. C++ standard library header files — Microsoft Docs. https://docs.microsoft.com/en-us/cpp/standard-library/cpp-standard-library-header-files?view=msvc-170. Accessed: 2022-04-06.

[Org] Catch Organization. Catch2. https://github.com/catchorg/Catch2. Accessed: 2022-04-07.

[OSK+11] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-preikschat. Octopos: A parallel operating system for invasive computing. *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA'11)*, page 9–14, 2011.

[Shu] Amit Shukla. Introduction to C++ STL (Standard Template Library). https://www.includehelp.com/stl/introduction-to-cpp-standard-template-library.aspx. Accessed: 2022-03-11.

[TAB+22] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Comp. Phys. Comm.*, 271:108171, 2022.

[THH+11] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive computing: An overview. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, pages 241–268, New York, NY, 2011. Springer New York.