



University  
of Glasgow | School of  
Computing Science

Honours Individual Project Dissertation

## LEVEL 4 PROJECT REPORT

**Zhang HanSheng**  
January 31th, 2019

# Abstract

This is a individual project that transforms NASA's Parallel Benchmark, also known as NPB from its original OpenMP based C version to C++ version which uses modern programming libraries like Intel's ParallelSTL or Khronos Group's triSYCL in order to conveniently test parallel computing performance across different hardware configurations and platforms using generic underlying industry standards like OpenCL/OpenGL/CUDA.

# Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Zhang HanSheng    Date: 01 February 2018

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation	1
1.2	Aim	1
1.3	Outline	2
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Concepts	3
2.1.1	OpenMP	3
2.1.2	ParallelSTL	4
2.2	OpenCL	5
2.3	SYCL	6
2.4	Libraries	6
2.4.1	triSYCL	6
2.4.2	C++ Range Library	8
2.5	Current Implementation of the NAS Parallel Benchmarks	8
<b>3</b>	<b>Analysis/Requirements</b>	<b>10</b>
3.1	Requirements Gathering	10
3.2	Previous Work	10
3.3	Functional Requirements	11
3.3.1	Must Have	11
3.3.2	Could Have	11
3.3.3	Should Have	12
3.3.4	Won't Have	12
<b>4</b>	<b>Design</b>	<b>13</b>
4.1	Subsection of benchmarks to implement	13
4.2	Design Choices	13
<b>5</b>	<b>Implementation</b>	<b>15</b>
5.1	BT	15
5.2	CG	17
5.3	range-v3 Extensions	18
5.4	Forcing SIMD Support	19
<b>6</b>	<b>Evaluation</b>	<b>21</b>
6.1	CPU	21
6.1.1	BT	22
6.1.2	CG	25
6.1.3	FT	28
6.2	Mobile CPU	31
6.3	GPU	32
6.4	Performance Analysis and tuning	32
6.5	Conclusion	40
6.6	Future Work	40

<b>Appendices</b>	<b>41</b>
<b>A Appendices</b>	<b>41</b>
A.1 Source Structure	41
A.2 Building	41
<b>Bibliography</b>	<b>42</b>

# 1 | Introduction

As Moore's Law rapidly reaches its last gasp, it has become even more obvious for computer scientists that relying on single core performance is unlikely to get much further and parallel computing is the de facto future for computation-heavy tasks. With technologies like Machine Learning and Deep Learning become an integral part of people's daily life. It's important to benchmark the computation units available in any computation scenario and dispatch computation tasks properly. This project attempts to port existing benchmark suites to a cross-platform implementation to satisfy this kind of needs. NASA's Parallel Benchmark is NASA's collection of a small set of programs that are derived from multiple real-world computation tasks like fluid dynamics, unstructured adaptive mesh and computational grids. As the time of writing this, the most widely accepted implementation is NPB3.0-omp-C, which uses OpenMP and thus provides very little compatibility according to OpenMP Architecture Review Boards. In this project, the aim is to rewrite this implementation with modern programming constructs to improve its compatibility in order to support all kinds of computation devices.

## 1.1 Motivation

Nowadays, multiple computation methods exist. There is traditional CPU computation that can be written in a wide range of languages of choice. However, as technology evolves, running computation on devices like a General Purpose Graphics Processing Unit has also gained some popularity. However, due to how GPUs are designed, programming on GPUs requires writing specialized so-called computation kernel which makes running legacy code on GPU a complicated task and needs to be done by hand.

## 1.2 Aim

The objective of this project is to investigate if it's possible to port the industry standard NASA's Parallel Benchmark from existing platform dependent non-portable OpenMP C implementation to C++ using modern programming constructs as well as OpenCL based triSYCL in order to measure the raw performance for both CPUs and GPUs. This could be split into two steps, the first being reimplementing the old version with C++ and Intel's ParallelSTL and ThreadBuildingBlock, the second being adapting ParallelSTL with GPU oriented triSYCL. A bonus attempt would be to convert the matrix operations in the aforementioned benchmark program into high-level optimized implementations through optimizing at the algorithm's level.

In order to achieve these objectives, the following key problems must be identified and addressed:

**Understanding** the old lengthy C implementation that contains many OpenMP pragmas and overused global variables as well as bad variable naming

**Understanding** OpenMP Pragmas and their meanings

**Understanding** the interface of the alternative C++ ParallelSTL interfaces so it's possible to switch between the two implementations

**Setting up** the environments properly including components like OpenMP enabled compilers as well as proper OpenCL related drivers so cross-hardware testing is possible

**Implementing** the benchmarks using ParallelSTL interfaces

**Researching** the status and current limitation of ParallelSTLs and the possibility of running the benchmark on GPUs with them.

**Evaluating** the performances of the newly implemented benchmarks on different devices.

## 1.3 Outline

In this work, I look at the steps taken to investigate whether it's possible to re-implement existing parallel computation benchmarks using C++ and thus provide more compatibility across different computation devices as well as actual benchmarking two implementations across a range of different devices. This includes constructing an OpenMP as well as a C++ demo to understand the basic concepts in both implementations, implementing a few benchmarks in Intel's ParallelSTL, implement those benchmarks in Khronos Group's variation triSYCL. I will also present what is STL, what's STL range and how modern range libraries like Ericniebler's range-v3 could help with improving the benchmark speed when used in conjunction with previously mentioned technologies. The remainder of this document is divided into the following chapters:

### Chapter 2 - Background and Related Work

This chapter focus on explaining current related works and introduce the C++ libraries that will be used in this project, as well as any potential knowledge needed.

### Chapter 3 - Requirements Analysis

This chapter analyzes the potential solutions to achieve the project's requirements

### Chapter 4 - Design

This chapter designs the benchmark, presents an overview of the libraries and technologies used as well as some discussion about the design choices

### Chapter 5 - Implementation

This chapter discusses the implementation of the benchmark, explaining the purpose of each variation as well as any issues encountered.

### Chapter 6 - Evaluation

This chapter discusses in great detail about how the implementation is tested and evaluated, as well as how different computing devices perform when using each of the variations.

### Chapter 7 - Conclusion

This chapter summarises the report and suggests some ideas for future research.

## 2 | Background

This chapter discusses current implementation and background knowledge needed

### 2.1 Concepts

#### 2.1.1 OpenMP

OpenMP is a multi-platform shared-memory parallel programming API that supports most desktop instruction set architectures and operating systems like Linux, macOS and Windows. OpenMP's implementation has two parts which we will explain how it works below: A standard compilant compiler as well as a runtime library. Due to those two requirements it has less compatibility when targeting newer platforms like iOS or Android which no official support exists. In its fairly newer specification OpenMP 4.5, OpenMP gained ability to offload computation tasks to other computation devices like GPU or specialized acceleration card. However as the time of writing, major compiler vendors like Clang/LLVM still have limited offloading device architecture support and usually focus on NVIDIA's CUDA API for GPU offloading. The current implementation of OpenMP has the compiler frontend first read through the pragmas specified by the programmer, transform it a bit then generated code with the actual computation combined with calls to functions in the OpenMP runtime library. For example, the following code creates threads running in parallel for vector multiplication.

```
#include <stdio.h>
int main(int argc, char **argv) {
    int ins[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
    int outs[16] = {0};
    #pragma omp parallel for
    for (int i = 0; i < 16; i++) {
        outs[i] = ins[i] * ins[i];
    }
    for (int i = 0; i < 16; i++) {
        printf("%d*d=%d\n", ins[i], ins[i], outs[i]);
    }
    return 0;
}
```

As we can see, OpenMP is fairly straightforward to use, however it still requires the programmer to specify attributes like shared and private variables in real world examples and thus could be troublesome and error-prone.



### 2.1.2 ParallelSTL

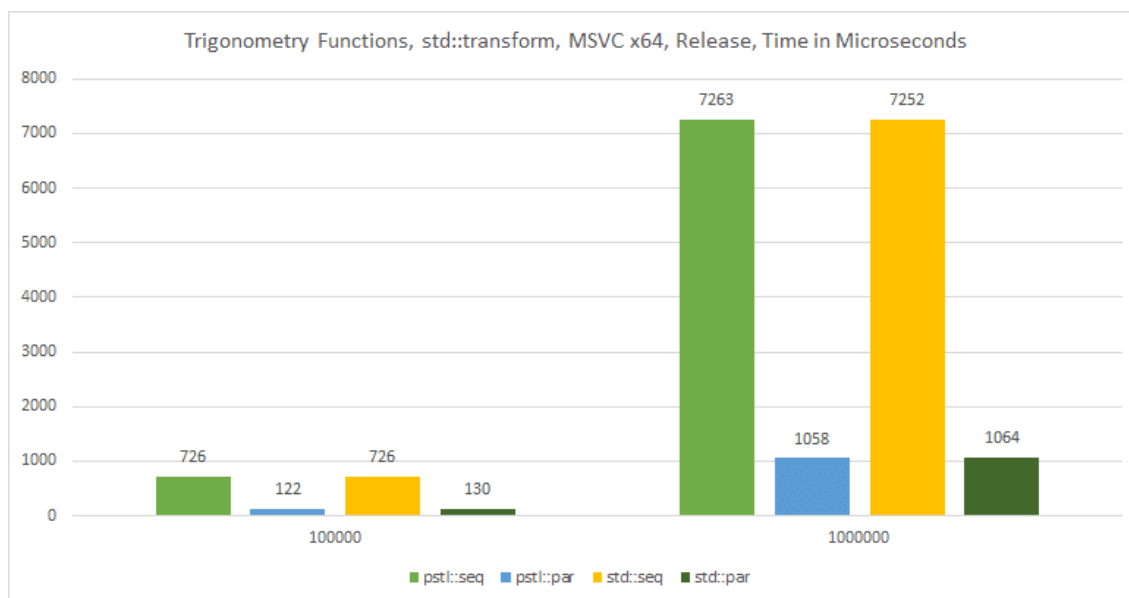
ParallelSTL could be used to reference both the C++ Parallelism Extension Standard as well as Intel's implementation of the C++ standard library which supports C++ 17 execution policies that could use a variety of parallel/threading solutions as its backend and has been proposed to be merged into LLVM's C++ STL by Alexey. Intel's implementation by default uses OpenMP4.0 or processor's native SIMD instruction as its SIMD execution backend by using `#pragma omp SIMD` directives which leaves the work of making the code running in SIMD (Also known as vectorization) to the compiler itself and thus does not require any OpenMP runtime library support. For other usages, Intel's implementation uses Intel's own ThreadBuildingBlock as its Parallel Execution Library, although in theory, it could also support other Parallel Backends. As the time of writing, Intel's implementation still has only CPU support. Intel has begun to contribute this library into GCC and Clang/LLVM's C++ standard library and thus can be expected to become more widely accepted in foreseeable future. Similar concepts of a parallel version of C++ STL have been widely adopted by hardware vendors and standard committees like NVIDIA/Khronos Group which all had their own implementation. ParallelSTL implementations provide a set of execution policies that specify if the code should run in parallel as well as SIMD attribution. An example of using ParallelSTL to copy one vector's elements into another is demonstrated below:

```
std::vector<int> vec = buildvectors(...);
std::vector<int> out(vec.size());
std::transform(std::execution::par, vec.begin(), vec.end(), out.begin(), [=](int
    i){
        return i*i;
    });
```

Each implementation might name the policies differently, but generally, ParallelSTLs provides the following execution policies:

- seq Sequential Execution
- unseq Unsequenced Parallel Execution
- par Multi-threaded Parallel execution
- par\_unseq Combined effect of unseq and par

Parallel execution is undoubtedly many times faster than the sequential version, as can be seen from Figure 2.1



**Figure 2.1:** Overview of the performance of the execution policies provided by Intel ParallelSTL. Credit:Filipek

However, the real benefit of such ParallelSTL implementations is that due to the unified interface that will soon move into the C++ standard, it's fairly easy to switch between ParallelSTL implementations to enable running the code on different devices, as we shall demonstrate later on into this document.

## 2.2 OpenCL

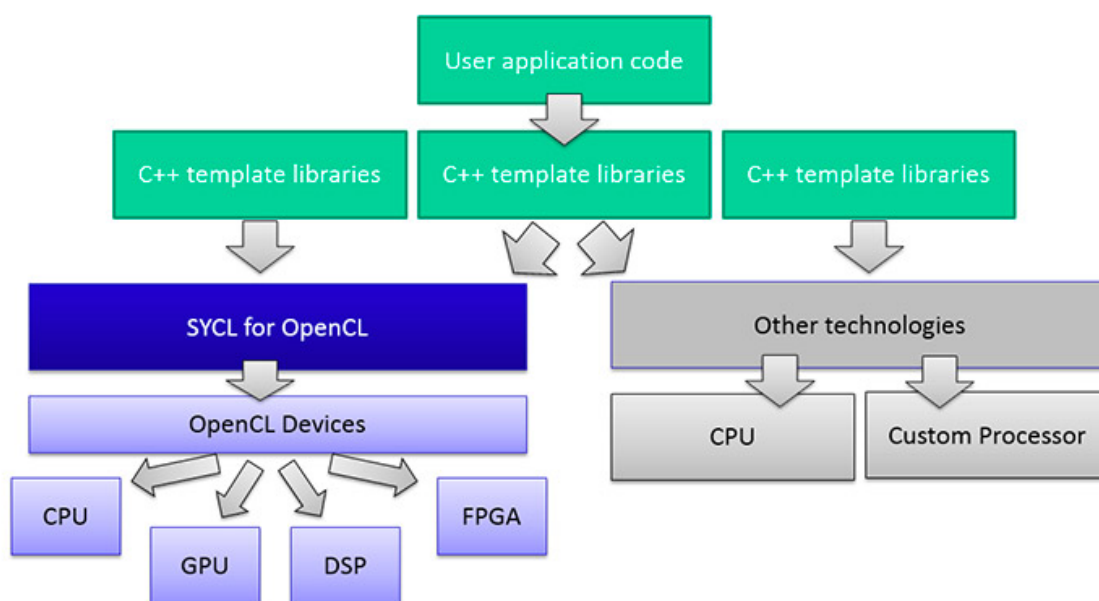
OpenCL is an open cross-platform parallel computation standard that supports a variety of devices ranging from CPUs and GPUs found in personal computers to embedded devices. It has its own computation kernel language which is a static subset of C++ 14. For example, the following OpenCL kernel code squares each element of an input array:

```
__kernel void square(__global float* i, __global float* ot, const unsigned int cnt){
    int idx = get_global_id(0); if(idx < cnt){
        ot[idx] = i[idx] * i[idx];
    }
}
```

On top of the OpenCL kernel code, the programmer also needs to maintain a client-side code which talks to the OpenCL driver and do tasks like allocating device buffer, setting up kernel arguments as well as retrieving the results. In this example, the management code has around 50 lines of code which is not shown here. This makes directly programming OpenCL code very troublesome and error-prone.

## 2.3 SYCL

SYCL is a C++ wrapper abstraction layer which uses OpenCL's concepts, portability and efficiency at its core to enable coding for heterogeneous processors using standard C++ in a single source file. It contains lambdas and templates to allow clean coding of high-level applications without optimized acceleration of kernel code. It still provides access to lower-level code through seamless integration of C++/C code as well as libraries and frameworks like OpenMP and OpenCL. As we can see below in 2.2, SYCL uses C++ template libraries to achieve "Write Once, Run Anywhere" of C++ computation code by providing a unified C++ level interface that's powered by a variety of SYCL-Compatible technologies like OpenCL and CPU.



*Figure 2.2: SYCL Overview. Khronos Group*

Listing 2.1 is a example slightly modified from Reyes to demonstrate how to square each element of an input array. This enables the programmer to write computation code in a single source file without having to worry too much about buffer allocation or transferring buffers between the device and the host.

## 2.4 Libraries

### 2.4.1 triSYCL

triSYCL is Khronos Group's implementation for its SYCL Heterogeneous Programming abstraction layer. It uses aforementioned Intel ParallelSTL and Intel ThreadBuildingBlocks for its CPU implementation, but it also adds the capability to use OpenCL to offload the computation tasks to any OpenCL device available in the system. Since OpenCL is widely adopted in mobile devices as well as desktops, this is the most widely compatible parallel computation library.

```

#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

constexpr size_t N = 3;
using Vector = float[N];

int main() {
    Vector a = { 1, 2, 3 };
    float c[N];

    { // By sticking all the SYCL work in a {} block, we ensure
      // all SYCL tasks must complete before exiting the block

        // Create a queue to work on
        queue myQueue;

        // Create buffers from a & b vectors with 2 different syntax
        buffer<float> A (std::begin(a), std::end(a));
        // A buffer of N float using the storage of c
        buffer<float> C(c, N);

        /* The command group describing all operations needed for the kernel
           execution */
        myQueue.submit([&](handler &cgh) {
            // In the kernel A and B are read, but C is written
            auto ka = A.get_access<access::mode::read>(cgh);
            auto kc = C.get_access<access::mode::write>(cgh);

            // Enqueue a single, simple task
            cgh.single_task<class sequential_vector>([=] () {
                for (size_t i = 0; i != N; i++)
                    kc[i] = ka[i] * ka[i];
            });
        }); // End of our commands for this queue
    } // End scope, so we wait for the queue to complete

    std::cout << "Result:" << std::endl;
    for (size_t i = 0; i != N; i++)
        std::cout << c[i] << " ";
    std::cout << std::endl;

    return 0;
}

```

*Listing 2.1: SYCL Example code for vector multiplication*

## 2.4.2 C++ Range Library

range-v3 is Eric Niebler's high-performance C++ 11/14/17 range library that has been introduced into C++ 20. It provides a C++ iterator which does lazy range computation and thus can be used to speed up computations that use ranges. For example, the following code constructs a lazily evaluated range starting from 0 and ends at 6:

```
#include <range/v3/all.hpp>
using namespace ranges;
.....
auto mrange = view::ints(0, 6);
```

Which could be written as the following using nowadays's STL like the following:

```
std::list<int> ls(6);
std::iota(ls.begin(), ls.end(), 0);
```

As one could notice, the STL version takes more memory and during the computation of `std::iota` more memory access is being made so should in theory slower than the range version. Plus, these ranges also have an iterator interface with functions like `::begin` and `::end` which makes them work with STL/ParallelSTL algorithms who take iterators out-of-the-box. Furthermore, range-v3 provides an easily extendable interface for the programmer to define a custom range (officially called view), which will be demonstrated later in this piece of work.

## 2.5 Current Implementation of the NAS Parallel Benchmarks

The current implementation used as a basis in this piece of work is **NAS Parallel Benchmarks 3.0 structured OpenMP C versions**, which is an unofficial implementation of the NPB based on top of the OpenMP API. It contains many OpenMP pragmas as well as meaningless variable names which makes it extremely difficult to understand. The C implementation also contains code snippet like the following, taken from the Block Tri-diagonal solver benchmark:

```
#pragma omp for
for (i = 0; i < grid_points[0]; i++) {
    for (j = 0; j < grid_points[1]; j++) {
        for (k = 0; k < grid_points[2]; k++) {
            for (m = 0; m < 5; m++) {
                for (n = 0; n < 5; n++) {
                    lhs[i][j][k][0][m][n] = 0.0;
                    lhs[i][j][k][1][m][n] = 0.0;
                    lhs[i][j][k][2][m][n] = 0.0;
                }
            }
        }
    }
}
```

This code snippet initializes the 1hs multi-dimension array by filling the array's inner elements with 0. This loop alone isn't really interesting, but it shows that the inner loops, especially the innermost one, could be further parallelised. Such design patterns and lack of parallelization could be seen in many places elsewhere inside the code, like the following, also taken from the same benchmark:

```
#pragma omp for
for (j = 1; j < grid_points[1] - 1; j++) {
    eta = (double)j * dnym1;
    for (k = 1; k < grid_points[2] - 1; k++) {
        zeta = (double)k * dnzm1;
        for (i = 0; i < grid_points[0]; i++) {
            xi = (double)i * dnxm1;
            exact_solution(xi, eta, zeta, dtemp);
            for (m = 0; m < 5; m++) {
                ue[i][m] = dtemp[m];
            }
            dtpp = 1.0 / dtemp[0];
            for (m = 1; m <= 4; m++) {
                buf[i][m] = dtpp * dtemp[m];
            }
            cuf[i] = buf[i][1] * buf[i][1];
            buf[i][0] = cuf[i] + buf[i][2] * buf[i][2] + buf[i][3] * buf[i][3];
        }
        for (i = 1; i < grid_points[0] - 1; i++) {
            im1 = i - 1;
        }
    }
}
```

## 3 | Analysis/Requirements

### 3.1 Requirements Gathering

I had multiple discussions with my supervisor Michel Steuwer who is responsible for leading me through the basic concepts as well as introducing the project itself. From those discussions I learnt about the main problems about the current implementation of the NPB benchmark, how will the new implementation be used and some of the issues I might encounter.

The requirements that got identified from those meetings are as following:

- Create a port of some of the old C/OpenMP based NPB benchmarks in C++
- Use modern technologies like ParallelSTL / triSYNC for cross-platform compiler-neutral compatible implementation
- Compare performances across different types of computing devices using the new implementation
- Compare the performance before and after using the lazy range library range-v3
- Implement a new programming construct on top of existing libraries
- Attempt to further optimize the benchmark using the new programming constructs

There were also some stretch goals

- Implement multiple benchmarks from the set
- The ability to optimize the benchmark at the algorithm's level instead of computation's level

### 3.2 Previous Work

When starting this project, it's useful to learn from previous attempts in this field in order to. Previously, my supervisor has done a high-performance parallel stencil code generator, LIFT. Stencil computation is a kind of computational kernel that updates elements in an array according to a fixed pattern. Stencil computation is very important because it could be found in many applications, especially computer simulation and image processing applications. LIFT has an extremely useful paper titled **High-Performance Stencil Code Generation with Lift** <http://www.lift-project.org/publications/2018/hagedorn18Stencils.pdf> that introduced the following stencil computation extensions as well as a concept called chained computation that allows multiple computation primitives to be chained together to form a new computation kernel. This design allows the expression of complex real-world computations like image blurring.

- pad which adds arbitrary padding elements to a sequence of unaligned data
- slide which as its name suggests, slides the sequence by an arbitrary offset, which is common in multi-dimension/stencil calculations
- map which applied a function to all the elements in a sequence and stores and returns values of the functions into a new array which has the same length as the old one

- iterate which applies a function for a certain amount of iterations and uses the output from the iteration as the input of its next iteration
- zip which creates a sequence of tuples from combining the two input sequences
- split splits the input sequence of equally sized chunks of n elements
- join which does of opposite of split

### 3.3 Functional Requirements

The MoSCoW method was used in order to document the requirements of the system. One of the key points of the MoSCoW method is to divide requirements by their importance. The following choices importance are available:

- Must have
- Should have
- Could have
- Won't have

Therefore, after careful consideration and discussion, the aforementioned requirements are divided into the following categories

#### 3.3.1 Must Have

These requirements must be satisfied

Requirement	Description
Port	Create a port of some of the old C/OpenMP based NPB benchmarks in C++
Modern Techniques	Use modern technologies like ParallelSTL and range-v3 library to allow easy portability to other computation platforms

#### 3.3.2 Could Have

These requirements would be great to have but are not exactly necessary

Requirement	Description
Optimize	Attempt to further optimize the benchmark using the new programming constructs
Algorithm	The ability to optimize the benchmark at the algorithm's level instead of computation's level
GPU	Run a few benchmarks on the GPU and compare the performance



### 3.3.3 Should Have

Requirement	Description
Compare across devices	Compare performances across different types of computation devices using the
Compare before and after range-v3	Compare the performance before and after using the lazy range library range-
Implement new function	Implement a new programming construct on top of existing libraries

### 3.3.4 Won't Have

These requirements are either overcomplicated or provides little benefit and thus will not be implemented

Requirement	Description
Implement the full benchmark set	Implement all the NPB benchmark programs

## 4 | Design

This chapter discusses the design of the C++ implementation in detail, presenting an overview of the new system's structure, the tools and libraries used as well as an in-depth explanation of the design choices.

### 4.1 Subsection of benchmarks to implement

The new structure of the benchmark system consists of three benchmarks stripped down from the C benchmark set listed below as suggested by my advisor. NAS Parallel Benchmarks 3.0 OpenMP C version by Popovt is used as the reference implementation.

- **BT** - Block Tri-diagonal solver
- **CG** - Conjugate Gradient, irregular memory access and communication
- **FT** - Discrete 3D fast Fourier Transform, all-to-all communication

### 4.2 Design Choices

When observing the original implementation, I noticed that those benchmarks support two modes known as pre-compiled data mode as well as loading data from user's argument. Since using pre-compiled data from the original implementation is more than enough to simplify the build process while keeping the benchmark's scientific correctness, I decided to go this way.

During the initial designing stage, I decided to use C++'s ParallelSTL, Clang/LLVM's Standard Library as well as OpenCL based triSYCL STL, the reason for this design is because all those libraries have similar interfaces so it's possible to toggle between three implementations at ease. Furthermore, in order to measure how lazy evaluation affects the memory footprint as well as the performance of those benchmarks, another build option is added to allow switching between range-v3's lazy evaluation library and C++ Standard Library's normal evaluation range library which generates an array containing the full possible value range beforehand. For example, the following code snippet generates a fully evaluated range of integers from -4 to 5 inclusively:

```
#include <algorithm>
#include <list>
#include <numeric>
.....
std::list<int> ls(10);
std::iota(ls.begin(), ls.end(), -4);
for(auto num: ls){
    std::cout << num << ' ';
}
std::cout << '\n';
```

```
.....
```

Which prints out the following:

```
-4 -3 -2 -1 0 1 2 3 4 5
```

Using range-v3 library, the same effect could be achieved with much less code, as demonstrated below:

```
auto ran=view::ints(-4, 6);
std::for_each(ran.begin(), ran.end(), [&](auto num) -> void {
    std::cout << num << ' ';
});
std::cout << '\n';
```

For the execution part, we mainly focus on Intel's ParallelSTL implementation, for example, the following code copies elements from the vector `from` to the vector `to` using C++ Standard Library's sequential `std::copy` implementation:

```
std::copy(from.begin(), from.end(), std::back_inserter(to));
```

While the following achieves the same affect using ParallelSTL's vectorized implementation:

```
#include "pstl/execution"
#include "pstl/algorithm"
std::copy(pstl::execution::unseq, from.begin(), from.end(), to.begin());
```

Since the project is focused on benchmarking the performance of scientific computations, the design also includes building some new programming constructs and libraries suitable for this purpose. In the LIFT project's paper, two new extensions, `slide` and `pad` is introduced and their purposes have been fully explained in previous parts of this work. Range-v3 provides a lazy evaluated new concept called `view`(combinators) which provides better composability when compared to the old style iterator based implementations. This allows the programmer to pass a range through an arbitrary amount of combinators which vastly improves the code's readability. Furthermore, the combinators could be programmed to be lazy evaluated or purely functional as well, which in theory allows even more performance boost. Due to these reasons, the design choice was made to attempt to implement those two extensions on top of range-v3's infrastructure by first subclassing `view_adaptor` and implement the aforementioned extensions.

## 5 | Implementation

In this chapter, the implementation of the new system is discussed in detail as well as an explanation, any issues encountered are also described. This section is outlined as follows:

- BT
- CG
- range-v3 Extensions
- Issues and Lessons Learnt During Implementation

### 5.1 BT

Block Tri-diagonal solver is the most complicated benchmark algorithm in the three selected benchmark sets, it has around 3600 lines of code and contains many long multi-layer-nested for-loops. However, the benchmark code snippets inside each code block are very similar and as such could be easily copied and pasted. The main issue in implementing this benchmark is to properly identify the critical section as well as the code blocks that could be paralleled. After reading the code, one could easily see a large amount of misused global variables that could result in a racing condition when executed in parallel. For example, consider the following code:

```
for (j = 1; j < grid_points[1] - 1; j++) {
    for (k = 1; k < grid_points[2] - 1; k++) {
        for (i = 0; i < grid_points[0]; i++) {
            tmp1 = 1.0 / u[i][j][k][0];
            tmp2 = tmp1 * tmp1;
            tmp3 = tmp1 * tmp2;
            /*many lines of code using these three variables*/
        }
    }
}
```

And these variables are declared in the header as `static double tmp1, tmp2, tmp3;` which makes parallel access to these variables result in undefined behaviour, so the first step was to remove the global declarations and update each code block referencing these variables to use a local declared temporary variable.

In the example above, the

```
for (j = 1; j < grid_points[1] - 1; j++) {
    for (k = 1; k < grid_points[2] - 1; k++) {
        for (i = 0; i < grid_points[0]; i++) {
            double tmp1 = 1.0 / u[i][j][k][0];
            double tmp2 = tmp1 * tmp1;
            double tmp3 = tmp1 * tmp2;
```

```

    /*many lines of code using these three variables*/
  }
}
}

```

Identifying SIMD sections is also a critical part in optimizing the performance, for example in the following code snippet, the inner-most for loop contains only SIMD operations and thus could use SIMD intrinsics to instruct the compiler to accelerate this part

```

irange = view::ints(1, grid_points[0] - 1);
jrange = view::ints(3, grid_points[1] - 3);
mrange = view::ints(0, 5);
krange = view::ints(1, grid_points[2] - 1);
std::for_each(PARALLEL, mrange.begin(), mrange.end(), [&](auto m) -> void {
    std::for_each(PARALLEL, krange.begin(), krange.end(), [&](auto k) -> void {
        std::for_each(PARALLEL, irange.begin(), irange.end(), [&](auto i) -> void {
            std::for_each(PARALLELUNSEQ, jrange.begin(), jrange.end(),
                [&](auto j) -> void {
                    rhs[i][j][k][m] =
                        rhs[i][j][k][m] -
                        dssp * (u[i][j - 2][k][m] - 4.0 * u[i][j - 1][k][m] +
                            6.0 * u[i][j][k][m] -
                            4.0 * u[i][j + 1][k][m] + u[i][j + 2][k][m]);
                });
        });
    });
});

```

## 5.2 CG

The Conjugate Gradient benchmark is the most complicated benchmark in the selected three benchmark suites due to its heavy use of more advanced OpenMP pragmas like `default(shared)` and `private(j)`

The first one means that by default all outer scope variables referenced by this OpenMP code block is shared among all threads, or each thread has its own copy if the pragma is defined as `default(private)`.

The second pragma serves as a complement to the aforementioned first part of the OpenMP pragma and specifies the thread-private variables if the first part has specified the default to be shared, and vice versa.

Most of the code blocks using aforementioned syntax are pretty straightforward and makes the extra OpenMP pragmas redundant. For example in the following code snippet:

```
#pragma omp parallel for default(shared) private(j)
for (j = 1; j <= lastcol - firstcol + 1; j++) {
    x[j] = norm_temp12 * z[j];
}
```

The only private variable in this code block is the loop index `j`, so it could be rewritten into the following format pretty straightforwardly:

```
auto jrange = view::ints(1, lastcol - firstcol + 2);
std::for_each(PARALLELUNSEQ, jrange.begin(), jrange.end(), [&](auto j) -> void {
    x[j] = norm_temp12 * z[j];
});
```

The reduction operators are unfortunately not directly supported by any of the ParallelSTLs so its effects are emulated using hand-written code built on top of `std::reduce`. This part is much more complicated as each code block needs to be analyzed and fully understood before re-implementation is possible. For example the following code block:

```
{
#pragma omp for reduction(+:rho)
for (j = 1; j <= lastcol-firstcol+1; j++) {
    rho = rho + r[j]*r[j];
}
}
```

Which takes the squared sum of each element in the range of `r`. This could be done in one line by using `std::accumulate` with a custom lambda like the following:

```
double rho =
    std::accumulate(&r[1], &r[lastcol - firstcol + 2], 0.0,
        [=](double i, double j) -> double { return i + j * j; });
```

### 5.3 range-v3 Extensions

Working with my supervisor, we also worked some extensions to the range-v3, for example the aforementioned `slide` concept. This was achieved by sub-classing the `view::view_adaptor` and override its `read`, `next`, `begin_adaptor` as well as its constructor. `Slide` is chosen because it's the most important and widely used operation in the concepts mentioned by the LIFT paper, while being fairly straightforward to implement.

Below is a sample of summing up the neighboring elements in a stencil using the extension:

```
#include "range.hpp"
#include <iostream>
#include <range/v3/all.hpp>
#include <string>
using namespace ranges;
int main(int argc, char const *argv[]) {
    auto t = view::ints(0, 10);
    std::cout << "Generated a range from 0 to 10:" << std::endl;
    copy(t, ostream_iterator<>(std::cout, " "));
    std::cout << "\nSliding with step=1 and size=3:" << std::endl;
    auto y = t | slide(1, 3);
    copy(y, ostream_iterator<>(std::cout, "\n"));
    ranges::for_each(y, [=](std::vector<int> num) {
        int val = std::accumulate(num.begin(), num.end(), 0);
        std::cout << "Sum of:[";
        copy(num, ostream_iterator<>(std::cout, " "));
        std::cout << "]" is: << val << "\n";
    });
    return 0;
}
```

Which prints out the following:

```
Generated a range from 0 to 10:
0 1 2 3 4 5 6 7 8 9
Sliding with step=1 and size=3:
[0,1,2]
[1,2,3]
[2,3,4]
[3,4,5]
[4,5,6]
[5,6,7]
[6,7,8]
[7,8,9]
[8,9]
[9]
Sum of:[0 1 2 ] is:3
Sum of:[1 2 3 ] is:6
Sum of:[2 3 4 ] is:9
Sum of:[3 4 5 ] is:12
Sum of:[4 5 6 ] is:15
Sum of:[5 6 7 ] is:18
Sum of:[6 7 8 ] is:21
Sum of:[7 8 9 ] is:24
Sum of:[8 9 ] is:17
```

```
Sum of:[9 ] is:9
```

Which can verify that this implementation is correct. However due to compatibility reasons for future GPU-enhancement, this implementation is not used in the main benchmark set even for CPU benchmarks

## 5.4 Forcing SIMD Support

The ParallelSTL can still uses OpenMP as its Parallel Backend, however the implementation of the execution policy `std::execution::par_unseq` and `std::execution::unseq` is controlled by a marco that is not fully correct, as the code snippet from `include/pstl/internal/pstl_config.h` shown below:

```
#if __clang__
// according to clang documentation, version can be vendor specific
#define __PSTL_CLANG_VERSION (__clang_major__ * 10000 + __clang_minor__ * 100 +
    __clang_patchlevel__)
#endif
/ Enable SIMD for compilers that support OpenMP 4.0
#if (_OPENMP >= 201307) || (__INTEL_COMPILER >= 1600) ||
    (!defined(__INTEL_COMPILER) && __PSTL_GCC_VERSION >= 40900)
#define __PSTL_PRAGMA_SIMD __PSTL_PRAGMA(omp simd)
#define __PSTL_PRAGMA_DECLARE_SIMD __PSTL_PRAGMA(omp declare simd)
#define __PSTL_PRAGMA_SIMD_REDUCTION(PRM) __PSTL_PRAGMA(omp simd reduction(PRM))
#elif !defined(_MSC_VER) //pragma simd
#define __PSTL_PRAGMA_SIMD __PSTL_PRAGMA(simd)
#define __PSTL_PRAGMA_DECLARE_SIMD
#define __PSTL_PRAGMA_SIMD_REDUCTION(PRM) __PSTL_PRAGMA(simd reduction(PRM))
#else //no simd
#define __PSTL_PRAGMA_SIMD
#define __PSTL_PRAGMA_DECLARE_SIMD
#define __PSTL_PRAGMA_SIMD_REDUCTION(PRM)
#endif //Enable SIMD
```

Which ignores Clang/LLVM compilers, while Clang/LLVM and GCC have had their own Auto Vectorizer implemented for a while, this feature is still heavily compiler vendor dependent and as such it's not suitable to depend the benchmark on a unstable compiler feature. For example GCC needs the user to explicitly pass compile flags to enable such vectorizer according to its website <https://www.gnu.org/software/gcc/projects/tree-ssa/vectorization.html>. As such in the testing environment the ParallelSTL is patched into the following to correctly explicitly state the SIMD attribute.

```
#if (_OPENMP >= 201307) || (__INTEL_COMPILER >= 1600) ||
    (!defined(__INTEL_COMPILER) && __PSTL_GCC_VERSION >= 40900) ||
    (__PSTL_APPLE_CLANG_VERSION > 8000038) || (__PSTL_CLANG_VERSION > 390000)
```

As shown in Figure 5.1, this SIMD modification is working as intended



The screenshot displays the BT.hop debugger interface. The main window shows assembly code with instructions and their corresponding memory addresses. The code includes various SIMD instructions like `movaps`, `movups`, `movabs`, and `mov` for `qword` and `xmmword` types. The right sidebar shows a tree view with sections like File, Local, CPU, Call, Address, Navigation, Instructions, and Graphs. The bottom status bar indicates the analysis progress and completion time.

```

BT.hop

mov     qword [qword_10115bcf0], 0x0          ; qword_10115bcf0
mov     qword [qword_10115bce8], 0x0          ; qword_10115bce8
movaps  xmm0, xmmword [0x1000113a0]          ; 0x1000113a0
movups  xmmword [qword_10115bcf0+8], xmm0     ; 0x10115bcf8
movaps  xmm0, xmmword [0x1000113b0]          ; 0x1000113b0
movups  xmmword [qword_10115bcf0+24], xmm0    ; 0x10115bd08
movaps  xmm0, xmmword [0x1000113c0]          ; 0x1000113c0
movups  xmmword [qword_10115bcf0+40], xmm0    ; 0x10115bd18
movaps  xmm0, xmmword [0x1000113d0]          ; 0x1000113d0
movups  xmmword [qword_10115bcf0+56], xmm0    ; 0x10115bd28
movaps  xmm0, xmmword [0x1000113e0]          ; 0x1000113e0
movups  xmmword [qword_10115bcf0+72], xmm0    ; 0x10115bd38
movabs  rbx, 0x3fff000000000000
mov     qword [qword_10115bd48], rbx          ; qword_10115bd48
xorps   xmm1, xmm1
movaps  xmmword [qword_10115bd48+8], xmm1     ; 0x10115bd50
mov     qword [qword_10115bd60], 0x0          ; qword_10115bd60
movaps  xmm2, xmmword [0x1000113f0]          ; 0x1000113f0
movups  xmmword [qword_10115bd60+8], xmm2     ; 0x10115bd68
movaps  xmm2, xmmword [0x100011400]          ; 0x100011400
movups  xmmword [qword_10115bd60+24], xmm2    ; 0x10115bd78
movaps  xmm2, xmmword [0x100011410]          ; 0x100011410
movups  xmmword [qword_10115bd60+40], xmm2    ; 0x10115bd88
movups  xmmword [qword_10115bd60+56], xmm0    ; 0x10115bd98
movaps  xmm0, xmmword [0x100011420]          ; 0x100011420
movups  xmmword [qword_10115bd60+72], xmm0    ; 0x10115bda8
mov     qword [qword_10115bdb8], rax          ; qword_10115bdb8
movaps  xmmword [qword_10115bdb8+8], xmm1     ; 0x10115bdc0
mov     qword [qword_10115bdd0], 0x0          ; qword_10115bdd0
movaps  xmm0, xmmword [0x100011430]          ; 0x100011430
movups  xmmword [qword_10115bdd0+8], xmm0     ; 0x10115bdd8
movaps  xmm1, xmmword [0x100011440]          ; 0x100011440
movups  xmmword [qword_10115bdd0+24], xmm1    ; 0x10115bde8
movaps  xmm1, xmmword [0x100011450]          ; 0x100011450
movups  xmmword [qword_10115bdd0+40], xmm1    ; 0x10115bdf8
movaps  xmm1, xmmword [0x100011460]          ; 0x100011460
movups  xmmword [qword_10115bdd0+56], xmm1    ; 0x10115be08
mov     qword [qword_10115be18], rax          ; qword_10115be18
mov     qword [qword_10115be20], rax          ; qword_10115be20
mov     qword [qword_10115be38], 0x0          ; qword_10115be38
mov     qword [qword_10115be30], 0x0          ; qword_10115be30
mov     qword [qword_10115be28], 0x0          ; qword_10115be28
movaps  xmmword [qword_10115be38+8], xmm0     ; 0x10115be40
movaps  xmm0, xmmword [0x100011470]          ; 0x100011470
movaps  xmmword [qword_10115be38+24], xmm0    ; 0x10115be50
movaps  xmm0, xmmword [0x100011480]          ; 0x100011480
movaps  xmmword [qword_10115be38+40], xmm0    ; 0x10115be60
movaps  xmm0, xmmword [0x100011490]          ; 0x100011490
movaps  xmmword [qword_10115be38+56], xmm0    ; 0x10115be70
movaps  xmm0, xmmword [0x1000114a0]          ; 0x1000114a0
movaps  xmmword [qword_10115be38+72], xmm0    ; 0x10115be80
movaps  xmm0, xmmword [0x1000114b0]          ; 0x1000114b0

> dataflow analysis of procedures in segment __LINKEDIT
> dataflow analysis of procedures in segment External Symbols
> Analysis pass 9/10: remaining prologs search
> Analysis pass 10/10: searching contiguous code area
> Last pass done
Background analysis ended in 497ms

>>> Python Command

```

ink in a new pane

Figure 5.1: SIMD attribute working as intended

## 6 | Evaluation

The evaluation was done by running the benchmark through various hardware/operating system/compiler configurations and compare the time taken. The benchmark is divided into three parts, consist of the following:

- **CPU** - Benchmark Results on a x86\_64 CPU
- **GPU** - Benchmark Results on a GPU
- **Mobile** - Benchmark Results on an Android Device
- **Analysis & Conclusion** - Analysis of the results on each of the tested hardware configurations as well as the conclusion drawn from them.

### 6.1 CPU

The CPU implementation's evaluation is done by using the same set of input data and compare the performance between the new C++ implementation against the old OpenMP C implementation on a MacBook Pro 15" Touch Bar Late 2016 with a 4Cores 8Threads 6-Gen Intel i7-6920HQ processor.

Both implementations are compiled with Clang/LLVM with -O3 optimization flag. Each C++ implementation is tested on two of the available Intel ParallelSTL's backends: OpenMP and Intel Thread Building Blocks. The version of related libraries and tools are listed below:

- **Clang/LLVM** - Clang/LLVM 7.0.0/Final x86\_64-apple-darwin18.2.0
- **libOpenMP** - LLVM 6.0.1 Stable
- **Intel ParallelSTL** - 20190305
- **Intel Thread Building Block** - TBB 4.4

Below are the benchmarks for each variation, both time spent and millions of operations per second are listed as well as their average value, then a graph is plotted to serve as the visual representation of the data as well as a basis to the final conclusion and analysis part. Note that during the implementation stage multiple pieces of research was done which resulted in improved performance, as noted in the Analysis section. These results are from the final implementation. Each benchmark is split into three variations, as explained below

- A** std::iota<> & Inner Loops not Parallel
- B** range-v3
- C** Nested Parallel Loops & std::iota
- D** Original C/OpenMP version as reference

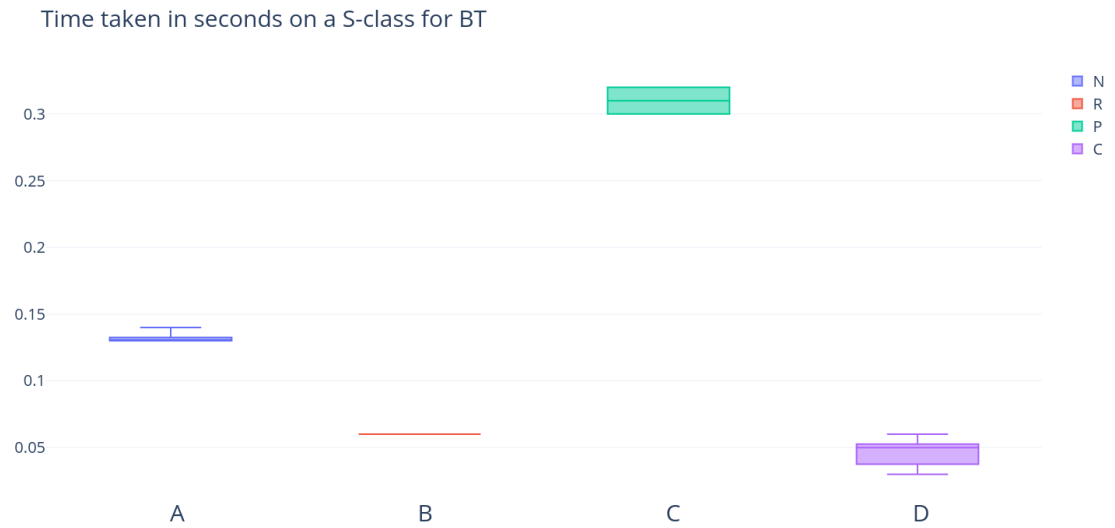
### 6.1.1 BT

BT (Block Tri-diagonal solver) is a nonlinear Partial differential equations solver that uses block tridiagonal algorithm. It has the following input data sets.

**BTS**

A	B	C	D
0.13	0.06	0.32	0.03
0.13	0.06	0.31	0.04
0.13	0.06	0.30	0.05
0.13	0.06	0.30	0.05
0.14	0.06	0.32	0.06

**Table 6.1:** Time for S-class



A	B	C	D
1702.88	3568.74	713.97	7290.41
1791.30	3739.29	731.69	5551.16
1729.94	3828.46	749.70	4561.17
1760.21	3660.58	750.30	4907.76
1657.59	3544.36	706.50	3889.37

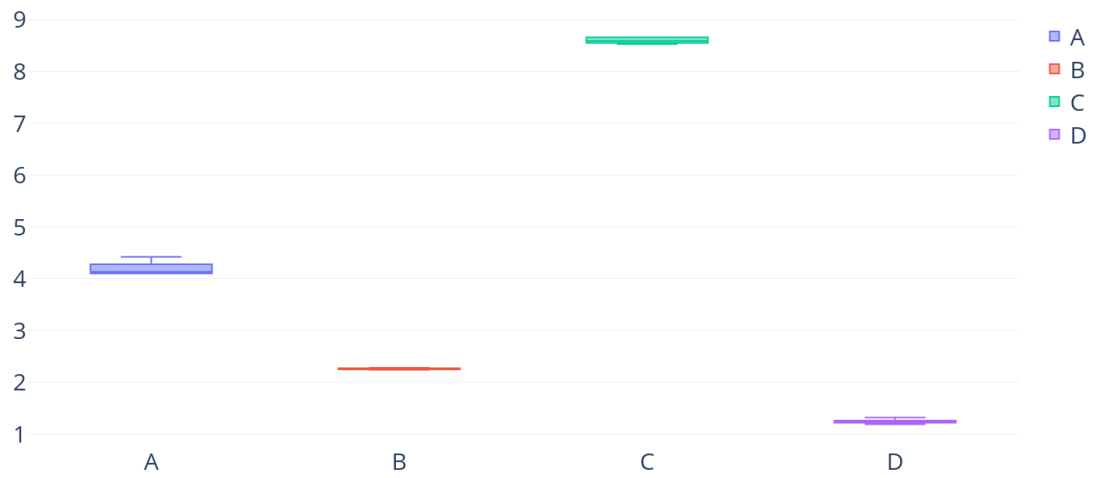
**Table 6.2:** Mop/s for S-class

**BTW**

A	B	C	D
4.10	2.24	8.66	1.32
4.13	2.25	8.56	1.24
4.10	2.27	8.53	1.19
4.23	2.27	8.59	1.24
4.42	2.28	8.66	1.23

**Table 6.3:** Time for W-class

Time taken in seconds on a W-class for BT



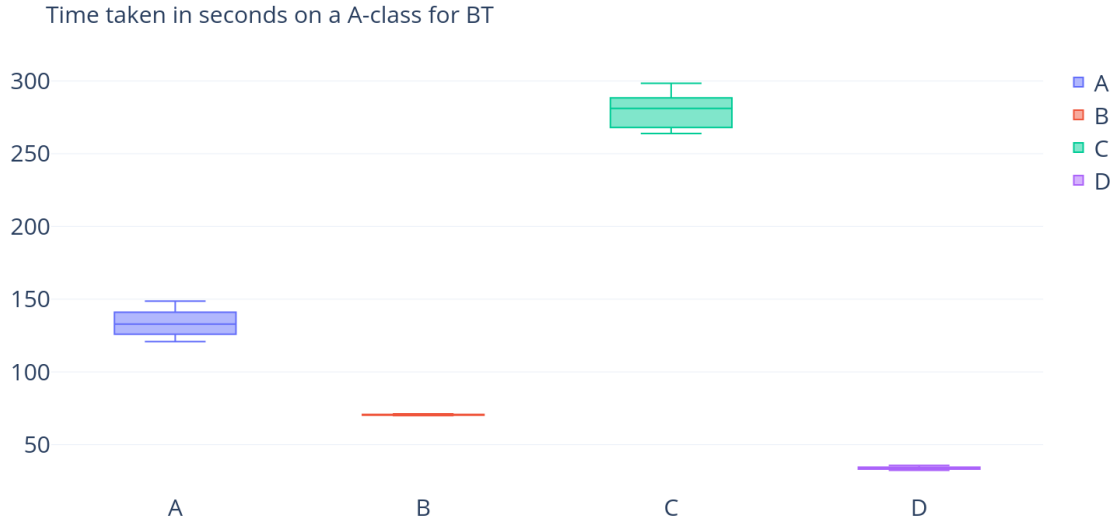
A	B	C	D
1883.08	3449.75	891.41	5865.59
1869.91	3437.97	901.69	6228.84
1884.47	3404.94	904.46	6471.26
1822.62	3399.71	898.98	6248.68
1746.36	3392.50	891.36	6284.49

**Table 6.4:** Mop/s for W-class

**BTA**

A	B	C	D
148.72	70.71	269.38	34.21
138.58	70.04	284.97	33.38
132.84	70.68	298.36	33.83
127.67	70.29	263.83	35.69
120.89	71.23	281.13	32.45

**Table 6.5:** Time for A-class



A	B	C	D
1131.53	2379.95	624.70	4919.32
1214.36	2402.58	590.53	5041.42
1266.79	2380.95	564.02	4974.79
1318.15	2394.07	637.85	4714.73
1392.09	2362.59	598.59	5186.15

**Table 6.6:** *Mop/s for A-class*

Overall, we can see that for BT, range-v3 slightly improves the performance of the three selected dataset when compared to `std::iota`. The parallelization of nested loops introduced a very heavy performance impact. Finally, OpenMP is significantly faster than other implementations.

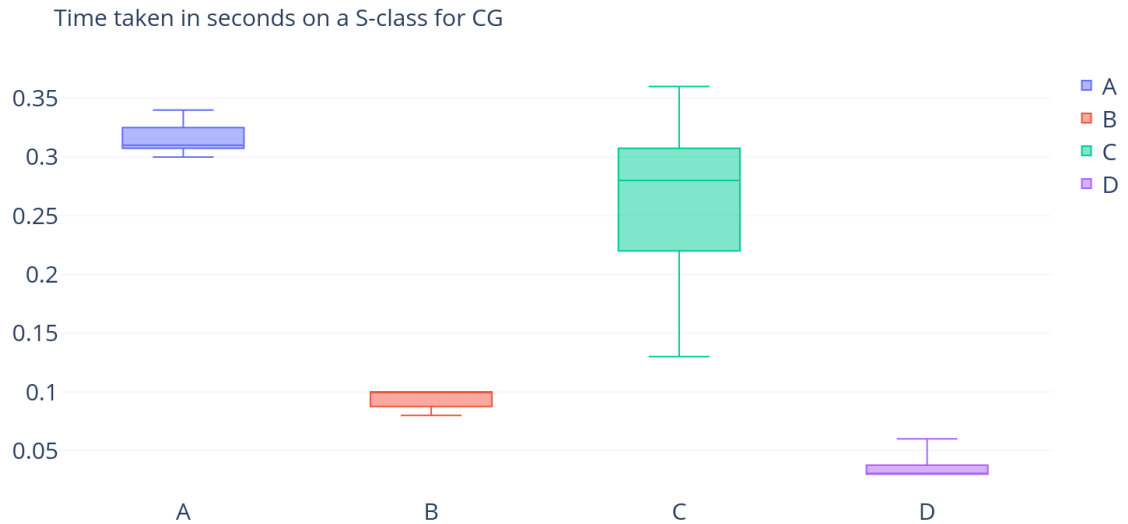
### 6.1.2 CG

CG is a linear equations system solver that uses the conjugate gradient method as a subroutine and implements the inverse iteration algorithm to estimate the smallest eigenvalue from a large sparse symmetric positive-definite matrix. It has the following input data sets.

**CGS**

A	B	C	D
0.30	0.10	0.13	0.06
0.31	0.10	0.25	0.03
0.31	0.09	0.29	0.03
0.34	0.10	0.28	0.03
0.32	0.08	0.36	0.03

**Table 6.7:** Time for S-class



A	B	C	D
225.40	669.14	220.36	1081.66
212.03	691.66	261.48	2525.16
217.49	720.50	226.85	2479.32
195.29	639.70	240.76	2559.48
206.90	821.02	184.45	2123.62

**Table 6.8:** Mop/s for S-class

**CGW**

A	B	C	D
1.37	0.40	1.34	0.08
1.41	0.49	1.25	0.06
1.33	0.40	1.28	0.06
1.44	0.42	1.24	0.07
1.41	0.47	1.27	0.10

**Table 6.9:** Time for W-class



A	B	C	D
306.48	1043.59	313.92	5092.68
297.99	862.52	335.77	6877.08
315.38	1057.11	329.79	6935.14
292.54	1004.34	339.36	5838.84
297.96	892.46	331.62	4216.50

**Table 6.10:** Mop/s for W-class

## CGA

A	B	C	D
3.50	1.48	3.19	0.35
3.47	1.50	3.22	0.38
3.49	1.55	3.19	0.41
3.52	1.53	3.17	0.36
3.54	1.53	3.25	0.38

**Table 6.11:** Time for A-class



A	B	C	D
427.14	1013.62	469.54	4321.92
431.32	998.38	465.40	3887.48
428.82	965.04	469.05	3628.95
425.08	975.01	471.65	4124.18
423.07	975.53	460.84	3977.61

**Table 6.12:** *Mop/s for A-class*

Overall, we can see that for CG, `std::iota` induced too much performance impact to an extent that its performance is comparable to the impact introduced by the parallelization of nested loops. Finally, OpenMP is significantly faster than other implementations.



### 6.1.3 FT

FT is a three-dimensional partial differential equation solver that implements the fast Fourier transform. **FTS**

A	B	C	D
0.09	0.10	0.73	0.04
0.10	0.10	0.72	0.03
0.10	0.10	0.76	0.04
0.10	0.10	0.72	0.04
0.11	0.10	0.73	0.03

**Table 6.13:** Time for S-class



A	B	C	D
1996.58	1721.49	241.56	4402.21
1775.42	1754.04	247.69	5661.73
1787.15	1770.85	234.60	5042.32
1786.53	1743.89	245.17	4681.63
1637.77	1747.67	243.92	5294.26

**Table 6.14:** Mop/s for S-class

**FTW**

A	B	C	D
0.23	0.24	1.62	0.07
0.23	0.25	1.42	0.07
0.23	0.25	1.52	0.07
0.23	0.25	1.44	0.07
0.24	0.26	1.47	0.07

**Table 6.15:** Time for W-class



A	B	C	D
1643.99	1542.88	229.93	5426.93
1621.67	1501.34	261.66	5462.57
1633.10	1485.52	244.97	5461.85
1637.60	1487.51	259.56	5280.59
1579.53	1414.62	253.29	5003.39

**Table 6.16:** Mop/s for W-class

## FTA

A	B	C	D
3.60	5.75	25.20	1.34
3.72	5.80	25.92	1.29
3.71	5.92	27.56	1.29
3.50	5.70	32.29	1.29
3.56	5.83	39.80	1.26

**Table 6.17:** Time for A-class



A	B	C	D
1980.96	1241.73	283.22	5314.95
1920.24	1230.70	275.29	5513.66
1922.25	1205.15	258.91	5529.14
2037.95	1251.10	221.01	5546.97
2005.84	1224.46	179.30	5642.96

**Table 6.18:** Mop/s for A-class

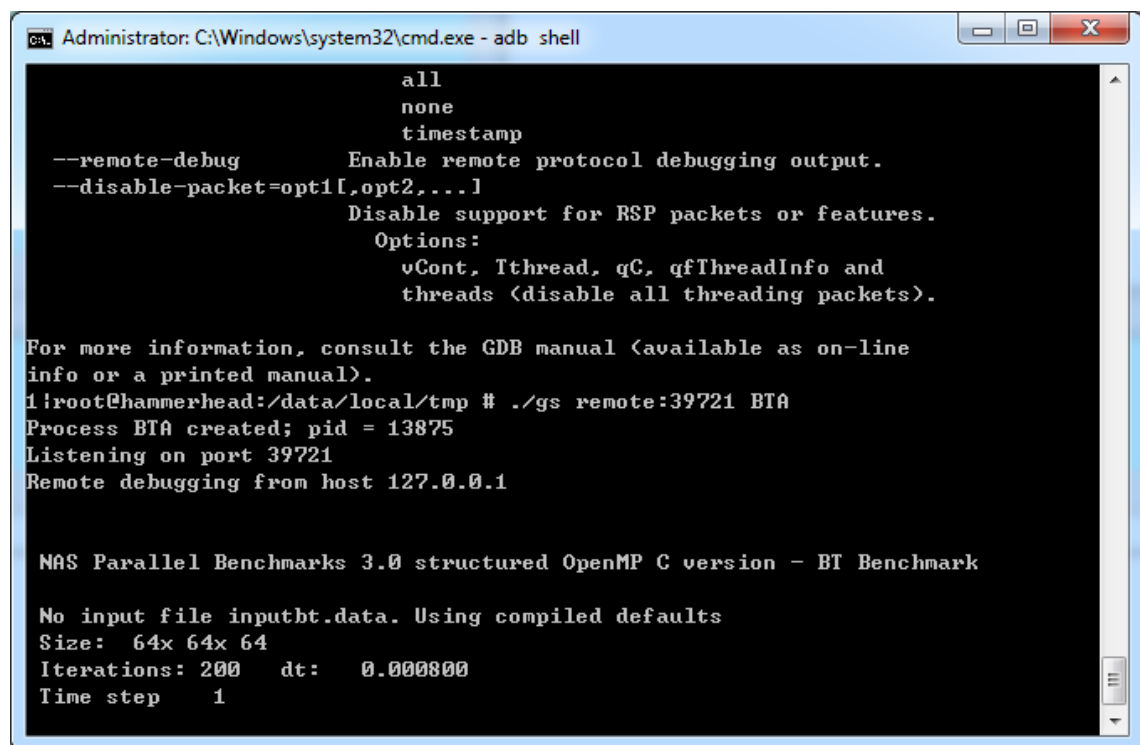
Overall, for FT, range-v3 is slower than the `std::iota` implementation and the performance impact increases as the input dataset grows in size. OpenMP is the fastest implementation. OpenMP is the fastest implementation.

## 6.2 Mobile CPU

The testing of mobile CPU is done on an Android 4.4 ARM-V7 device. Due to the nature of limited RAM and computation horsepower, only the smallest two subsets, S and A, are chosen to run. In practice, the rest larger subsets have all crashed for unknown reasons. Versions of the used libraries/environments are listed below:

- **Clang/LLVM** - Android (4751641 based on r328903)clang version 7.0.2 (<https://android.googlesource.com/toolchain/llvm> 003100370607242ddd5815e4a043907ea9004281)(<https://android.googlesource.com/toolchain/llvm> 1d739ffb0366421d383e04ff80ec2ee591315116)(based on LLVM 7.0.2svn)x86\_64-apple-darwin18.2.0
- **libOpenMP** - Unknown
- **Intel ParallelSTL** - 20190305
- **Intel Thread Building Block** - TBB 4.4

The mobile implementation in Intel TBB can be executable properly for the smaller input datasets like S,W, however for larger datasets the benchmark programmes crash randomly possibly due to the limited RAM. Below is a screenshot of BTA benchmarking running on an Android device:



```

Administrator: C:\Windows\system32\cmd.exe - adb shell

all
none
timestamp
--remote-debug      Enable remote protocol debugging output.
--disable-packet=opt1[,opt2,...]
                    Disable support for RSP packets or features.
                    Options:
                    vCont, Ithread, qC, qfThreadInfo and
                    threads <disable all threading packets>.

For more information, consult the GDB manual <available as on-line
info or a printed manual>.
1!root@hammerhead:/data/local/tmp # ./gs remote:39721 BTA
Process BTA created; pid = 13875
Listening on port 39721
Remote debugging from host 127.0.0.1

NAS Parallel Benchmarks 3.0 structured OpenMP C version - BT Benchmark

No input file inputbt.data. Using compiled defaults
Size:  64x 64x 64
Iterations: 200  dt:  0.000800
Time step  1

```

Figure 6.1: Screenshot.

Note that although the screenshot still says OpenMP C version, this screenshot is actually running the C++ ParallelSTL implementation. However, the execution attempts mostly result in overheat device, crash for no reason or unexpected device shutdown. For the very few attempts with the smaller datasets that are actually successful, the results are not interesting at all and either show similar trends to the PC version or the difference is too small due to the small dataset and thus provide no actual research meaning. As such those values are not attached.

## 6.3 GPU

Initially, the plan was to benchmark the GPU implementation on an Nvidia RTX 2080Ti with Khronos Group's SyclParallelSTL. However, this task was later proven to be impossible because SYCL and its implementations follow another binary GPU shader specification called SPIR-V, which is not supported by Nvidia's proprietary drivers as the time of writing. There are other ParallelSTL implementations similar to Khronos' SyclParallelSTL that supports CUDA. Thrust <https://thrust.github.com>, for example, is one of those CUDA-accelerated ParallelSTL implementations. However, yet another new issue was raised, as shown in its source code, that similar to SyclParallelSTL, Thrust provides only support for 2D arrays (otherwise known as vectors), which is not suitable for this project's use scenario. Initially, the plan was to implement part of the benchmarks that only uses 1-D arrays into GPU STL implementations. However, this plan was also proven to be unrealistic later on as potential candidates for this task still require multiple layers of indexing. However, since all of the major GPU shading languages (OpenCL/GLSL/CUDA/SPIR-V/Metal) supports Vector3 and higher dimension arrays natively or supports such data structure from allowing direct access to buffer pointers as shown previously in the OpenCL example, we can expect ParallelSTLs exposing such feature in foreseeable future.

## 6.4 Performance Analysis and tuning

Due to the limited data of mobile CPUs, the analysis part will focus on desktop implementations.

Initially, an assumption of the performance difference was caused by the function calling overhead was made since in BT the performance loss percentage dropped from 10% to 7% as the input data set size grows. While such assumption seems possible for a debug-mode build, it has been soon proven wrong in the release-mode build which was used to produce the benchmarks above, as those explicit functions are optimized out either through inlining or other means, as demonstrated below:

```
A : >>> nm -U release/BT |grep pstl | head -n 5
naville @ navilledesMacBook-Pro in ~/Desktop/L4Project (master••)
A : >>> nm -U debug/BT |grep pstl | head -n 5
0000000010000ddb0 t __ZN6__pstl8internal11brick_walk1IN6ranges2v314basic_iteratorINS3_9iota_viewIivEEEEZ10error_normPdE3$_2EEvT$_$
tegral_constantIbLb0EEE
0000000010000df30 t __ZN6__pstl8internal11brick_walk1IN6ranges2v314basic_iteratorINS3_9iota_viewIivEEEEZ10error_normPdE3$_3EEvT$_$
tegral_constantIbLb0EEE
0000000010001a440 t __ZN6__pstl8internal11brick_walk1IN6ranges2v314basic_iteratorINS3_9iota_viewIivEEEEZ10initializevE4$_13EEvT$_$
tegral_constantIbLb0EEE
0000000010001ae20 t __ZN6__pstl8internal11brick_walk1IN6ranges2v314basic_iteratorINS3_9iota_viewIivEEEEZ10initializevE4$_14EEvT$_$
tegral_constantIbLb0EEE
0000000010001bc00 t __ZN6__pstl8internal11brick_walk1IN6ranges2v314basic_iteratorINS3_9iota_viewIivEEEEZ10initializevE4$_15EEvT$_$
tegral_constantIbLb0EEE
```

*Figure 6.2: No STL template functions exists in a -O3 release build*

However, FT provides an compile marco called TIMERS\_ENABLED which allows us to print out the timer for each benchmark stage, which should enable better insight on exactly which part of the computation was responsible for the 50% slow down. So another round of benchmark using FT and Class A was carried out. The results are listed below:

	C++ OpenMP	C
Setup	0.542755	0.545380
FFT	2.620628	2.632424
Evolve	2.711872	0.143807
Checksum	0.000277	0.000246
Total	5.875539	3.321863

**Table 6.19:** Average Time taken in seconds

It can be seen that for other benchmark stages, the C++ implementation's performance is very close to its C/OpenMP counterpart, sometimes even slightly faster. However, the evolve stage is almost 20 times slower than its C counterpart. The following code snippet is responsible for timing the evolve stage:

```
if (TIMERS_ENABLED == TRUE) {
    timer_start(T_EVOLVE);
}

evolve(u0, u1, iter, indexmap, dims[0]);

if (TIMERS_ENABLED == TRUE) {
    timer_stop(T_EVOLVE);
}
```

Which suggests only the implementation of the function evolve affects the time spend for this part, but how is this method implemented?

Below is the C++ implementation code:

```
void evolve(dcomplex u0[NZ][NY][NX], dcomplex u1[NZ][NY][NX], int t,
            int indexmap[NZ][NY][NX], int d[3]) {
    to irange = view::ints(0, d[0]);
    to jrange = view::ints(0, d[1]);
    to krange = view::ints(0, d[2]);
    d::for_each(PARALLEL, irange.begin(), irange.end(), [&](auto i) -> void {
        std::for_each(PARALLEL, jrange.begin(), jrange.end(), [&](auto j) -> void {
            std::for_each(PARALLELUNSEQ, krange.begin(), krange.end(), [&](auto k) -> void {
                crmul(u1[k][j][i], u0[k][j][i], ex[t * indexmap[k][j][i]]);
            });
        });
    });
};
```

And the C implementation code:

```
void evolve(dcomplex u0[NZ][NY][NX], dcomplex u1[NZ][NY][NX],
            int t, int indexmap[NZ][NY][NX], int d[3]) {

    /*-----
    c-----*/

    /*-----
    c evolve u0 -> u1 (t time steps) in fourier space
```

```

c-----*/

    int i, j, k;

#pragma omp parallel for default(shared) private(i,j,k)
    for (k = 0; k < d[2]; k++) {
        for (j = 0; j < d[1]; j++) {
            for (i = 0; i < d[0]; i++) {
                crmul(u1[k][j][i], u0[k][j][i], ex[t*indexmap[k][j][i]]);
            }
        }
    }
}

```

Since `crmul` is just a simple compile-time macro defined as `#define crmul(c,a,b)(c.real = a.real * b, c.imag = a.imag * b)`, these two pieces of code could be summarized into three nested for-loops. Due to the simplicity of this piece of code, it serves as a perfect suite to test the performance between the ParallelSTL and raw OpenMP code, as no heavy computation or complex library calls are involved, the caller of this function is also relatively straightforward, as demonstrated above. In order to compare the generated code of two implementations, a disassembler, IDA Pro, is used and the following is the pseudocode generated for two implementations:

```

__int64 __fastcall evolve(__int64 a1, __int64 a2, int a3, __int64 a4, unsigned int *a5)
{
    __int64 result; // rax
    __int64 v6; // r10
    __int64 v7; // r14
    __int64 v8; // r14
    __int64 v9; // r8
    __int64 v10; // r15
    __int64 v11; // r13
    __int64 v12; // rdx
    __int64 v13; // r12
    __int64 v14; // rbx
    __int64 v15; // [rsp+0h] [rbp-30h]

    result = *a5;
    v15 = *a5;
    if ( *a5 )
    {
        v6 = a5[1];
        if ( (_DWORD)v6 )
        {
            v7 = a5[2];
            if ( (_DWORD)v7 )
            {
                v8 = v7 << 18;
                v9 = 0LL;
                v10 = a3;
                do
                {
                    v11 = a1;
                    result = a4;
                    v12 = a2;
                    v13 = 0LL;
                    do
                    {
                        v14 = 0LL;
                        do
                        {
                            *(__m128d *) (v12 + 4 * v14) = _mm_mul_pd(
                                _mm_loadup_pd((const double *)&v9 + v10 * *(signed int *) (result + v14)),
                                *(__m128d *) (v11 + 4 * v14));
                            v14 += 0x40000LL;
                        } while ( v8 != v14 );
                        ++v13;
                        v12 += 4096LL;
                        result += 1024LL;
                        v11 += 4096LL;
                    } while ( v13 != v6 );
                    ++v9;
                    a2 += 16LL;
                    a4 += 4LL;
                    a1 += 16LL;
                } while ( v9 != v15 );
            }
        }
    }
    return result;
}

```

Figure 6.3: C++ OpenMP version



```

1 __int64 __fastcall evolve(__int64 a1, __int64 a2, int a3, __int64 a4, signed int *a5)
2 {
3     __int64 result; // rax
4     __int64 v6; // r10
5     __int64 v7; // r14
6     __int64 v8; // rcx
7     __int64 v9; // r15
8     __int64 v10; // r13
9     __int64 v11; // rdx
10    __int64 v12; // r12
11    __int64 v13; // rbx
12    __int64 v14; // r9
13    __int64 v15; // [rsp+0h] [rbp-38h]
14    __int64 v16; // [rsp+8h] [rbp-30h]
15
16    v16 = a4;
17    result = a5[2];
18    v15 = result;
19    if ( result > 0 )
20    {
21        v6 = a5[1];
22        if ( v6 > 0 )
23        {
24            v7 = *a5;
25            v8 = 0LL;
26            v9 = a3;
27            do
28            {
29                if ( (signed int)v7 > 0 )
30                {
31                    v10 = a2;
32                    result = v16;
33                    v11 = a1;
34                    v12 = 0LL;
35                    do
36                    {
37                        v13 = 0LL;
38                        v14 = 0LL;
39                        do
40                        {
41                            *(__m128d *)(v10 + v13) = _mm_mul_pd(
42                                _mm_loaddup_pd((const double *)&v7 * *(signed int *)(result + 4 * v14++))
43                                *(__m128d *)(v11 + v13));
44                            v13 += 16LL;
45                        } while ( v14 < v7 );
46                        ++v12;
47                        v11 += 4096LL;
48                        result += 1024LL;
49                        v10 += 4096LL;
50                    } while ( v12 < v6 );
51                }
52                ++v8;
53                a1 += 0x1000000LL;
54                v16 += 0x400000LL;
55                a2 += 0x1000000LL;
56            } while ( v8 < v15 );
57        }
58    }
59    return result;
60 }

```

Figure 6.4: C/OpenMP version

As we can see, the core computation code inside the innermost scope looks almost the same on both implementations, as such we can deduce the performance drop is a result of one or more reasons listed below:

- range-v3 Library
- ParallelSTL
- The large amount of threads created, since each layer of the **for** loop has been parallelized.

In order to analyze if/how these possibilities affect the performance, two variations of the original code snippet was created as shown below:

```

//This tests the impact of the range-v3 library
// No Ranges Implementation
void evolve(dcomplex u0[NZ][NY][NX], dcomplex u1[NZ][NY][NX], int t,
           int indexmap[NZ][NY][NX], int d[3]) {
    std::vector<int> irange(d[0]);
    std::iota(irange.begin(), irange.end(), 0);
    std::vector<int> jrange(d[1]);
    std::iota(jrange.begin(), jrange.end(), 0);

```

```

std::vector<int> krange(d[2]);
std::iota(krange.begin(),krange.end(),0);
std::for_each(PARALLEL,irange.begin(), irange.end(), [&](auto i) -> void {
    std::for_each(PARALLEL,jrange.begin(), jrange.end(), [&](auto j) -> void {
        std::for_each(PARALLELUNSEQ,krange.begin(), krange.end(), [&](auto k) -> void
        {
            crmul(u1[k][j][i], u0[k][j][i], ex[t * indexmap[k][j][i]]);
        });
    });
});
}

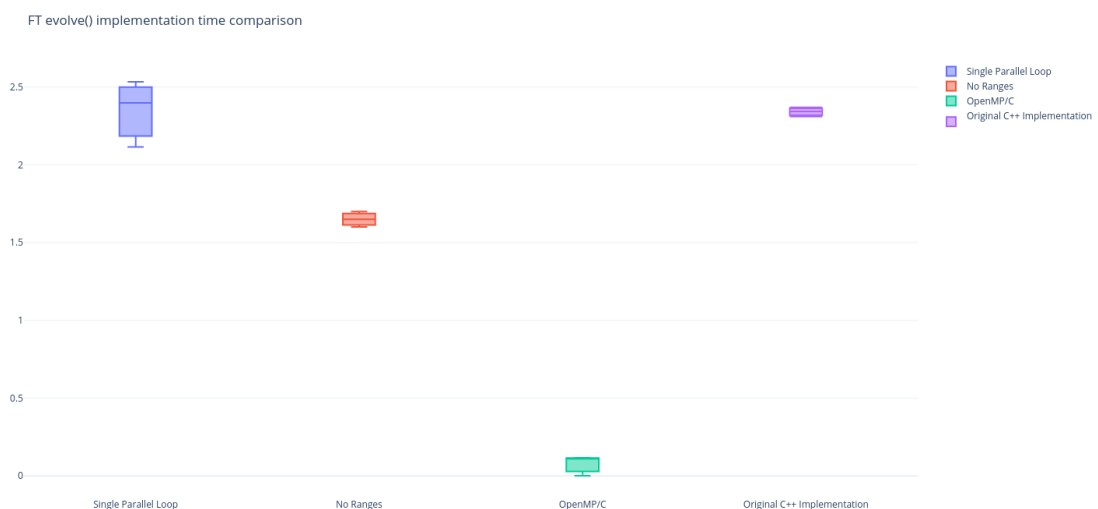
```

```

//This tests the impact of the threads
//Single Parallel Loop Implementation
void evolve(dcomplex u0[NZ][NY][NX], dcomplex u1[NZ][NY][NX], int t,
            int indexmap[NZ][NY][NX], int d[3]) {
    auto irange = view::ints(0, d[0]);
    auto jrange = view::ints(0, d[1]);
    auto krange = view::ints(0, d[2]);
    std::for_each(PARALLEL,irange.begin(), irange.end(), [&](auto i) -> void {
        std::for_each(jrange.begin(), jrange.end(), [&](auto j) -> void {
            std::for_each(krange.begin(), krange.end(), [&](auto k) -> void {
                crmul(u1[k][j][i], u0[k][j][i], ex[t * indexmap[k][j][i]]);
            });
        });
    });
}

```

Below is the graph of two implementations. From left to right the plots represent Single Parallel Loop, No Ranges, Original C/OpenMP Implementation and Original C++ Implementation from [this](#) project respectively.



*Figure 6.5: Time in seconds for each of the variations.*

Initially, when observing from the graph, the initial assumption is that threading is not the main reason for the performance impact due to how rewriting range-v3 with `std::iota` improves the performance significantly. However, later when comparing the binaries, I noticed that the binary doesn't link to Intel ParallelSTL's tbb execution backend when ranges library is used, which means that when using ranges library, the code is actually running in sequential order, which explains the huge performance drawback. As demonstrated below, symbols from the TBB library don't exist in the final product:

```

_ZN3tbb10interface78internal20isolate_within_arenaERNS1_13delegate_baseE1
_ZN3tbb18task_group_context4initEv
_ZN3tbb18task_group_contextD1Ev
_ZN3tbb4task13note_affinityEt
_ZN3tbb8internal36get_initial_auto_partitioner_divisorEv
_ZNK3tbb18task_group_context28is_group_execution_cancelledEv
_ZNK3tbb8internal120allocate_child_proxy8allocateEm
_ZNK3tbb8internal127allocate_continuation_proxy8allocateEm
_ZNK3tbb8internal132allocate_root_with_context_proxy4freeERNS_4taskE
_ZNK3tbb8internal132allocate_root_with_context_proxy8allocateEm
_ZNKSt3__120__vector_base_common11b1EE20__throw_length_errorEv
_ZSt9terminatev
_ZTIIN3tbb4taskE
_ZTIIS9bad_alloc
_ZTVN10__cxxabiv117__class_type_infoE
_ZTVN10__cxxabiv120__si_class_type_infoE
_ZTVN10__cxxabiv121__vmi_class_type_infoE
_ZdlPv
_Znwm
__bzero
__cxa_begin_catch
__cxa_end_catch
__cxa_rethrow
__gxx_personality_v0
__sincos_stret
__stack_chk_fail
__stack_chk_guard
__exit
__gettimeofday
__memset_pattern16
__printf
__puts
dyld_stub_binder
naville @ navilledeMacBook-Pro in ~/Development/L4Project/release (master•)
λ : >>> make FT
[ 66%] Built target Core
Scanning dependencies of target FT
[ 83%] Building CXX object CMakeFiles/FT.dir/FT/ft.cpp.o
[100%] Linking CXX executable FT
[100%] Built target FT
naville @ navilledeMacBook-Pro in ~/Development/L4Project/release (master•)
λ : >>> nm -u FT
__ZSt9terminatev
__cxa_begin_catch
__gxx_personality_v0
__sincos_stret
__stack_chk_fail
__stack_chk_guard
__exit
__gettimeofday
__memset_pattern16
__printf
__puts
dyld_stub_binder
naville @ navilledeMacBook-Pro in ~/Development/L4Project/release (master•)
λ : >>>

```

This is further proven by running the BTA benchmark in range-v3 mode and `std::iota` mode, as demonstrated in Figure ?? and Figure ??

```

Processes: 466 total, 4 running, 462 sleeping, 2187 threads
Load Avg: 4.79, 2.64, 2.21 CPU usage: 91.48% user, 5.95% sys, 2.56% idle SharedLibs: 322M resident, 63M data, 97M linkedit.
MemRegions: 232150 total, 7088M resident, 153M private, 1779M shared. PhysMem: 16G used (2999M wired), 81M unused.
VM: 6676G vsize, 1299M framework vsize, 6557243(0) swapins, 7481360(0) swapouts.
Networks: packets: 45167086/41G in, 33735279/5141M out. Disks: 48845361/1367G read, 43158973/653G written.
15:37:16

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID STATE BOOSTS %CPU_ME %CPU_OTHRS UID FAULTS COW
11873 BTA 624.7 02:02.52 8/8 0 273 300M 0B 0B 11873 11346 running *0[1] 0.00000 0.00000 501 76909 116

```

*Figure 6.6: B*  
TA's CPU usage is 627.4% when range-v3 is not used

```

Processes: 466 total, 4 running, 462 sleeping, 2128 threads
Load Avg: 2.29, 2.05, 2.00 CPU usage: 15.67% user, 3.31% sys, 81.0% idle SharedLibs: 322M resident, 63M data, 97M linkedit.
MemRegions: 231226 total, 7091M resident, 153M private, 1696M shared. PhysMem: 16G used (3004M wired), 83M unused.
VM: 6676G vsize, 1299M framework vsize, 6557115(0) swapins, 7481360(0) swapouts.
Networks: packets: 45166270/41G in, 33734720/5141M out. Disks: 48842278/1367G read, 43157549/653G written.
15:36:17

PID COMMAND %CPU TIME #TH #WQ #POR MEM PURG CMPR PGRP PPID STATE BOOSTS %CPU_ME %CPU_OTHRS UID FAULTS COW
11584 BTA 99.7 00:48.32 1/1 0 10 299M 0B 0B 11584 11346 running *0[1] 0.00000 0.00000 501 76874 108

```

*Figure 6.7: B*  
TA's CPU usage is 99.7% when range-v3 is used

Then in order to fully test the impact of the aforementioned observation, a marco called `MAKE_RANGE` and its variation that doesn't declare a new variable `MAKE_RANGE_UNDEF` is created and used. These two macros switch to `std::iota` instead of `range-v3` when required. Also, due to the current limitation of the `ParallelSTL`, this `std::iota` section is fully in sequential, which caused yet another performance downgrade, consider the following code:

```

MAKE_RANGE(0, 3, irange);
NS::for_each(PARALLEL, irange.begin(), irange.end(),
             [&](auto i) -> void { logd[i] = ilog2(d[i]); });
MAKE_RANGE(0, d[1], jrange);
NS::for_each(PARALLEL, jrange.begin(), jrange.end(), [&](auto j) -> void {
    cfftz(is, logd[2], d[2], y0, y1);
    MAKE_RANGE_UNDEF(0, d[2], krange);
    MAKE_RANGE_UNDEF(0, fftblock, irange);
    NS::for_each(PARALLEL, krange.begin(), krange.end(), [&](auto k) -> void {
        NS::for_each(PARALLELUNSEQ, irange.begin(), irange.end(), [&](auto i) -> void {
            {
                xout[k][j][i + ii].real = y0[k][i].real;
                xout[k][j][i + ii].imag = y0[k][i].imag;
            }
        });
    });
});
});

```

In this example, the ranges are created inside the loop when they could be created outside of the loop. While this doesn't create too much performance downgrade in ranges library as those ranges are lazy evaluated and thus has almost no cost to re-create them, this will cause serious performance impact when `USE_RANGES` marco is not defined and thus using the `std::iota` implementation because this would cause the vector to be created and looped inside every single loop. After this analysis, the codes are reviewed again to make sure such ranges are created at the outermost scope to avoid pointless recreation of non-lazy ranges.

Finally, the impact of the aforementioned assumption that creating too many threads also downgrade the performance is also evaluated and compared, by using the following macro I was able to control if the inner loops of an already paralleled loop are running in parallel as well:

```
#ifndef NONESTPARALLEL
#define NESTPAR SEQ
#define NESTPARUNSEQ UNSEQ
#else
#define NESTPAR PARALLEL
#define NESTPARUNSEQ PARALLELUNSEQ
#endif
```

The results of these variations are listed in 6.1. Overall, OpenMP is still the fastest implementation despite has low portability, which is something can be expected to improve when the aforementioned OpenMP GPU Offloading feature is fully implemented. For range-v3 and `std::iota`, as we can see from the results above, the lazy evaluated range library can be 2 to 3 times faster when compared to its non-lazy replacement `std::iota` while also using considerably less memory. However, the biggest drawback of range-v3 is that using it effectively removes most of the parallel computation support because today's ParallelSTL implementations don't support this kind of usage. Although as the ranges concept becomes standardized in C++20 it can be expected that such support is added to ParallelSTL implementations.

## 6.5 Conclusion

While porting the C version into C++ is feasible with the help of various ParallelSTL implementations, there left a lot more to be desired. The code quality of each ParallelSTL's implementation varies. The limitations of each STL is also worth mentioning. For example, modern GPU shading languages like GLSL, CUDA and SYCL all support high dimension data types like Vector3 and Vector4, but none of the ParallelSTLs explored in this project supports those data types, which is why it wasn't possible to run the benchmarks on GPU yet.

Although ParallelSTL as a relatively newborn concept still has a lot of work to do like supporting multi-dimensional arrays which already exists in modern GPU Shading languages like GLSL, CUDA and SYCL as well as OpenCL kernel code. The concept itself is indeed very useful. As we've demonstrated in this project, this extra abstraction layer allows the programmer to use a variety of execution policies and execution backends to run computation code with very little to none extra changes required.

## 6.6 Future Work

ParallelSTL's capabilities are restricted to compute a chosen code snippet the programmer has explicitly states. This introduces extra overhead like buffer copying for Heterogeneous computing. Another better approach would be using specialized compilers that compile the full source file into GPU Shading Language's format, either in its high-level language form or in its serialized format. This is feasible since Nvidia already has its LLVM based GPU compiler that reads in a subset of C programming language called CUDA C and emits NVPTX assembly.

# A | Appendices

## A.1 Source Structure

The source code folder contains the following folders:

**include/** Headers

**lib/** Shared code routines like timers

**CG/** Implementation code for CG

**BT/** Implementation code for FT

**r3test/** Demonstration code for the range-v3 extension code in `include/range.hpp`

## A.2 Building

Building requires cmake as well as Intel TBB to be installed. CMake is available at <https://cmake.org> or the software package manager in your system. Intel TBB is available at <https://github.com/01org/tbb/releases> or the software package manager of your choice. Although if software package manager is not used then CMake might have problems finding the library.

The following CMake options are allowed:

**USE\_RANGES** Uses range-v3 instead of `std::iota`

**NESTED\_PAR** Allows inner loops of nested for-loops to run in parallel

You can specify those by adding arguments to CMake in this format `-D OPTION=VALUE`, note that the accepted values for those two options are `ON` or `OFF`. There are a few extra options as well but at the current stage setting them to other values is meaningless and as thus they have been set to the suitable default value

## 6 | Bibliography

- K. Alexey. Proposal to contribute Intel implementation of C++17 parallel algorithms. URL <http://lists.llvm.org/pipermail/cfe-dev/2017-November/056135.html>.
- B. Filipek. How to boost performance with Intel Parallel STL and C++17 Parallel Algorithms. URL <https://dzone.com/articles/how-to-boost-performance-with-intel-parallel-stl-a>.
- Khronos Group. SYCL Overview. URL <https://www.khronos.org/sycl/>.
- OpenMP Architecture Review Boards. Openmp compilers & tools. URL <https://www.openmp.org/resources/openmp-compilers-tools/>.
- M. Popovt. NAS Parallel Benchmarks 3.0 structured OpenMP C versions. URL <https://github.com/benchmark-subsetting/NPB3.0-omp-C/>.
- R. Reyes. Sycl Tutorial 1: The Vector Addition. URL <https://www.codeplay.com/portal/sycl-tutorial-1-the-vector-addition>.