Term Report - Independent Study on MASS C++ and HPX

Rishabh Pratap Singh

Term Report for Independent Study (Su25)

submitted in partial fulfillment of the

requirements of the degree of

Master of Science in Computer Science & Software Engineering (or Cybersecurity Engineering)

University of Washington

08/22/2025

Supervisor:

Dr. Munehiro Fukuda

Abstract

This report summarizes a two-pronged independent study performed as preparatory work for a master's thesis on extending and modernizing **MASS C++**. The first strand focused on understanding MASS C++ internals, updating the codebase to work with modern dependencies, fixing security/usability issues, and making the library benchmarkable. The second strand involved installing and evaluating **HPX** to learn how its modern parallel/async features could inform proposed improvements to MASS C++. This document describes the motivation, implementation changes (with placeholders for code screenshots), recommended experimental procedures, and guidance for drafting the results and conclusion sections for the final report.

Table of Contents

1. Introduction

This independent study had two complementary objectives:

1. **MASS C++ modernization:** obtain a deep working knowledge of MASS C++, make pragmatic updates so the library builds and runs with modern toolchains, harden security for remote execution, and verify correctness by running sample programs and benchmarks.
2. **HPX exploration:** install HPX and its dependencies, run canonical benchmarks (Fibonacci and matrix multiplication), and analyse how HPX implements and exposes asynchronous and distributed execution patterns. Insights from HPX will inform design choices for thesis work aiming to extend MASS C++ with more modern concurrency and programmability features.

The outcomes of this term work are intended to:

- Ensure a functional, documented baseline for MASS C++.
- Provide comparative experimental data and implementation patterns from HPX.
- Obtain a concrete roadmap for the thesis effort.

2. Motivation

The motivating reasons for this study are:

- **Foundation for the thesis:** before proposing design or algorithmic changes at thesis scale, the codebase must be buildable, testable, and benchmarkable on current systems.
- **Learn modern concurrency patterns:** HPX's approach to futures, asynchronous actions, and parallel algorithms provides concrete implementations and APIs that can inspire safer, higher-level abstractions for MASS C++.
- **Practical improvements:** addressing compatibility, security, and usability issues in MASS C++ will reduce friction for future experiments and collaborators.
- **Benchmark-driven validation:** empirically validate that the modernized MASS C++ still meets expected performance and to quantify where HPX features could yield gains.

Key research questions framed by this study:

- What are the main portability/security/usability problems in the current MASS C++ codebase, and how can they be fixed without breaking API compatibility?
- Which HPX features map naturally to use-cases in MASS C++ (asynchronous remote execution, task graphs, parallel algorithms)?

- How do performance and programmability trade-offs compare between MASS C++ and HPX for representative workloads?

3. Implementation

3.1 MASS C++

- **libssh2 upgrade**
  - *Problem:* Existing libssh2 (v1.4.2) was incompatible with the system OpenSSH version (Open SSL 3.2.2 as seen in *Figure 3.1*), causing build/runtime failures for remote launch / SSH interactions (required libcrypto.so.10, which was unavailable).
  - *Action:* Upgraded libssh2 to v1.11.1(Which adds support for OpenSSL 3 as seen in *Figure 3.2*) and updated build scripts / CMake (or autotools) configuration to locate new headers and libraries. Adjusted any API call sites that changed between versions.
  - *Testing:* Rebuilt the project; exercised remote-launch code paths and verified successful SSH handshakes and command execution.

```
checking for inline... inline
checking non-blocking sockets style... O_NONBLOCK
configure: ERROR: No openssl crypto library found!
No libgcrypt crypto library found!
```

Figure 3.1.1: Libcrypto required dependency not found

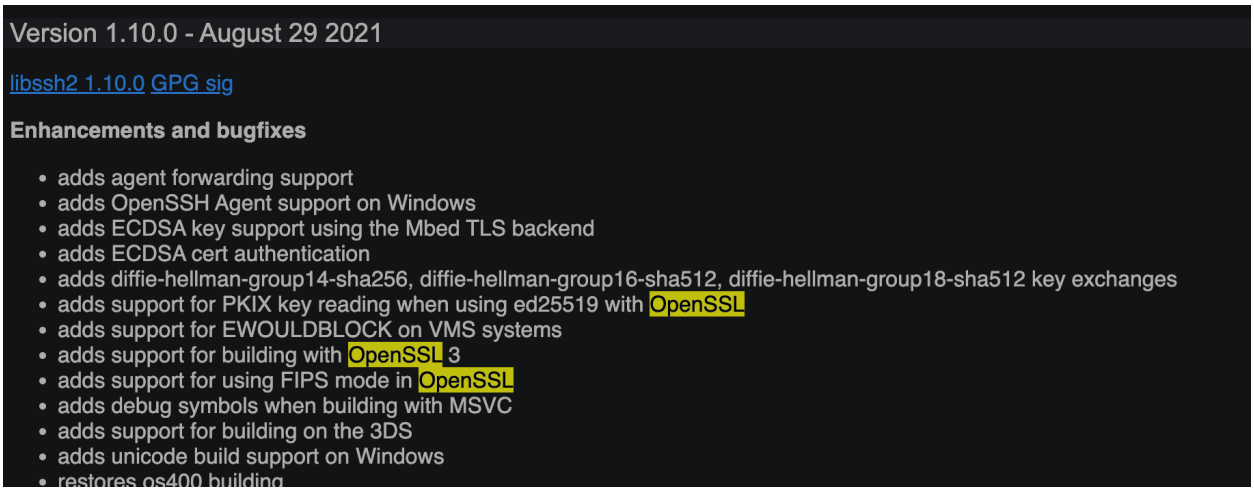

Figure 3.1.2: Libss2 update to support OpenSSL 3

- **Use public-key authentication by default**
    - *Problem:* The project defaulted to storing user passwords in plain text for remote connections, which is a security risk.
    - *Action:* Changed the remote-connection workflow to prefer public-key authentication (as seen in *Figure 3.3)*. Removed or refactored code paths that saved plain-text passwords; added logic to fallback to password only if explicitly provided by the user or configured in a secure credential manager.
    - *Testing:* Verified both password-less public-key authentication and explicit passphrase-based fallbacks. Confirmed no password strings are written to disk in normal operation.

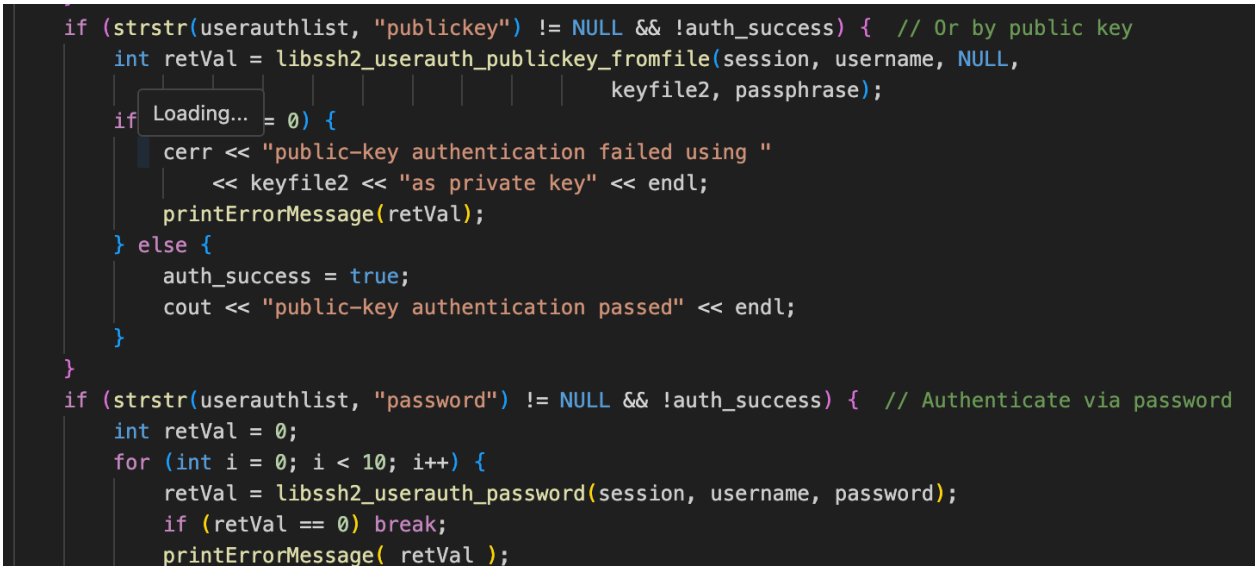

*Figure 3.1.*3: Connection attempted using public key before explicit password

- **ssh-agent not available for remote connection**
    - *Problem:* ssh-agents that are required to store user passwords and passphrases for repeated remote login were not initialized by default, which prevented subsequent connections.
    - *Action:* Added a script that checks if the service is going to have access to an ssh-agent, if yes then no ssh-agent is created otherwise the script creates a ssh-agent that MASS can use for creating sockets with remote machines without relying on explicit passwords.
    - *Testing:* Verified no errors were generated due to the lack of ssh-agents and all socket connections were able to successfully connect to remote machines.

```
# Check if SSH_AUTH_SOCK is set (indicating an ssh-agent is running)
if [ -z "$SSH_AUTH_SOCK" ]; then
    echo "SSH agent not running. Starting new agent..."
    # Start ssh-agent and evaluate its output to set environment variables
    eval "$(ssh-agent -s)"
    echo "SSH agent started with PID: $SSH_AGENT_PID"
else
    echo "SSH agent is already running."
fi
```

*Figure 3.1.*4: Starting an ssh-agent for the service if one doesn't already exist

- **Hostname verification without reverse DNS**
  - *Problem:* Reverse DNS lookups were used to identify incoming connections; unreliable or missing reverse DNS caused failures.
  - *Action:* Modified the connection handshake so the *remote* side includes its hostname in the initial message for identity verification. Retain optional reverse-DNS as a fallback, but do not depend on it. Added input validation to mitigate spoofing risks (e.g., require the remote-provided hostname to match other expected values or be validated against a configured mapping).
  - *Testing:* Verified connection establishment in environments lacking reverse DNS. Documented the handshake protocol change and compatibility implications.

```
// accept a new connection
int sd = socket->getServerSocket();

// Reading Server Hostname
int size=-1;

read( sd, (void *)&size, sizeof( int ) );
if (printOutput) {
            convert.str("");
            convert << "Server Hostname size is: " << size;
            MASS_base::log(convert.str());
        }
char *serverName = new char[size];
if (size > 0){
    // Reading the actual server name from the socket
    read( sd, serverName, size );
    if (printOutput) {
            convert.str("");
            convert << "Server Hostname: " << serverName << " whose size is: " << size;
            MASS_base::log(convert.str());
        }
}else{
    if (printOutput) {
            convert.str("");
            convert << "Failed to get server hostname";
            MASS_base::log(convert.str());
        }
    exit(-1);
}
```

*Figure 3.1.*5: When receiving a connection, wait for server's hostname

```
// Fetching hostname of the server and calculating it's size
std::string hostname = getHostName();
int msg_size = strlen(hostname.c_str());
if (printOutput) {
        convert.str("");
        convert << "Hostname is: " << hostname << " whose size is: " << size;
        MASS_base::log(convert.str());
}
// sends connection requests to lower ranks
for (int i = 0; i < rank; i++) {
    sockets[i] = socket->getClientSocket(hosts[i].c_str());

    // Sending the hostname from the server
    write(sockets[i], (void *)&msg_size, sizeof(int));
    write(sockets[i],hostname.c_str(),msg_size);
    fsync(sockets[i]);
```

Figure 3.1.6: Server hostname is sent after connection is established

- **Sample program updates**
    - *Problem:* Example code used outdated APIs or paths and failed to demonstrate remote execution end-to-end.
    - *Action:* Updated sample programs to compile and exercise both Place and Agent remote execution paths, with clear steps to reproduce. Added comments and debug statements that demonstrates how to run the example both locally and across multiple hosts.
    - *Testing:* Executed the sample across remote nodes and captured output verifying correct behaviour.

```
Places *land = new Places( 1, "Land", 1, msg, 7, 2, 100, 100 );

// change message to good
msg = (char *)("good\0");

// call the Land class's implementation of callAll
// with the msg and the length of the message.
land->callAll( Land::init_, msg, 5 );

// Create and populate and array of ints to represent locations.
```

Figure 3.1.7: Place testing across multiple nodes

```
//    handle, className, *argument, argument_size, *places,  initPopulation
Agents *nomad = new Agents( 2, "Nomad", msg, 7, land, 10000 );

// Perform the Agent's callAll with the
// agent implementation's function, msg, and message size.
// In this case it is an initialization routine.
nomad->callAll( Nomad::agentInit_, msg, 5 );
printf("Nomad has been initiated");

//Test callAll second time, this time on the Nomads, which are agents.
msg = (char *)("Second attempt\0");
nomad->callAll( Nomad::somethingFun_, msg, 15 );
```

Figure 3.1.8: Agent testing across multiple nodes

3.2 HPX

Summary of actions and artifacts to include

- **Dependency installation**
  - Installed/build prerequisites typically required by HPX:
    - Boost (1.88.0)
    - hwloc (2.12.1) for topology awareness
    - optional memory allocators (tcmalloc/jemalloc) for improved memory performance.
- **HPX build & install**
  - Download HPX source from Github.
  - Build using CMake utilizing available flags for customization.
  - Use makefile to do a selective build or complete build.
  - Some of the important flags provided by hpx installer:
    - -DHwloc_ROOT: To specify the root of hwloc installation.
    - -DBoost_ROOT: To specify the root of boost installation.
    - -DHPX_WITH_MALLOC: Custom or system memory manager.
    - -DHPX_WITH_FETCH_ASIO: If Asio needs to be installed.
    - -DCMAKE_INSTALL_PREFIX: To define hpx installation directory.
  - Detailed steps can be found here: Google Drive.
- **Benchmark preparation**
  - Implemented two canonical benchmarks:
    - **Fibonacci (recursive async):** demonstrates task creation, futures, and asynchronous recursion.

```cpp
//[fib_func
std::uint64_t fibonacci(std::uint64_t n)
{
    if (n < 2)
        return n;

    // We restrict ourselves to execute the Fibonacci function locally.
    hpx::id_type const locality_id = hpx::find_here();

    // Invoking the Fibonacci algorithm twice is inefficient.
    // However, we intentionally demonstrate it this way to create some
    // heavy workload.

    fibonacci_action fib;
    hpx::future<std::uint64_t> n1 = hpx::async(fib, locality_id, n - 1);
    hpx::future<std::uint64_t> n2 = hpx::async(fib, locality_id, n - 2);

    return n1.get() +
        n2.get();    // wait for the Futures to return their values
}
```

Figure 3.2.1: Fibonacci implementation with futures implementation

- **Matrix multiplication (parallel algorithm):** demonstrates data-parallel constructs and workload distribution.

```cpp
// Matrices have random values in the range [lower, upper]
std::uniform_int_distribution<element_type> dis(lower, upper);
auto generator = std::bind(dis, gen);
hpx::ranges::generate(A, generator);
hpx::ranges::generate(B, generator);

// Perform matrix multiplication
hpx::experimental::for_loop(hpx::execution::par, 0, rowsA, [&](auto i) {
    hpx::experimental::for_loop(0, colsB, [&](auto j) {
        R[i * colsR + j] = 0;
        hpx::experimental::for_loop(0, rowsB, [&](auto k) {
            R[i * colsR + j] += A[i * colsA + k] * B[k * colsB + j];
        });
    });
});
```

Figure 3.2.2: Matrix multiplication for parallel computing (with excellent programmability)

- **Async overhead:** Calculate overhead caused by async by waiting for each async call.

```cpp
    for (std::size_t i = 0; i < N; i++)
    {
        hpx::future<int> f = hpx::async(&null_task);
        f.get();
    }
```

Figure 3.2.3: Async overhead calculated by forcing linear operation

    - Each benchmark should have:
        - A brief description of what it measures (task granularity, async overhead, data movement costs).
        - The HPX implementation approach (e.g., hpx::async, hpx::parallel::for_loop, and hpx::actions).
- **Notable findings during execution:**
    - Hpx uses hpx::find_here() to find available resources available on the system/cluster, if fewer found than required HPX fails pre-emptively.
    - Hpx is modular by nature, it allows linking specific libraries within HPX according to requirements, HPX::hpx and HPX::wrap_main are mandatory.
    - Similar to agents, hpx has actions() which are asynchronous functions that hop to available resources for execution.

4. Results

- Successful end-to-end execution of Mass C++, with all of its remote and local features tested including callAll(), manageAll(), exchangeBoundary(), exchangeAll() for both Places and Agents. Verifying a successful update of required libraries and improvements to get Mass C++ upto date.

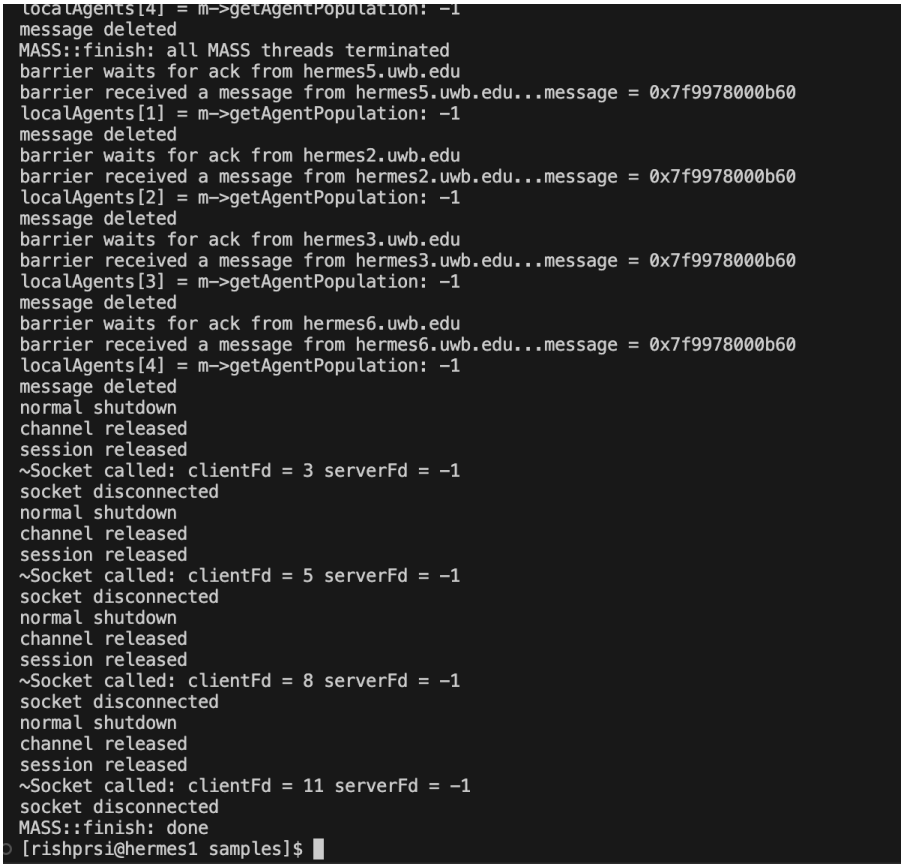

Figure 5.1: MASS C++ successful execution

- The following patterns were analyzed from the hpx benchmarks:
    - Matrix Multiplication: Close to linear scaling due to very high parallelizable logic of matrix multiplication.
    - Fibonacci: Constant performance with minimal benefit on even number of nodes, due to critical sections of the logic.
    - Parallel for: Good scaling because of simplicity of parallelizing for loops using hpx::for_each.
    - Async overhead: Running non parallel code on multiple nodes to single out overhead caused by async. Which increases with the increase in number of nodes.

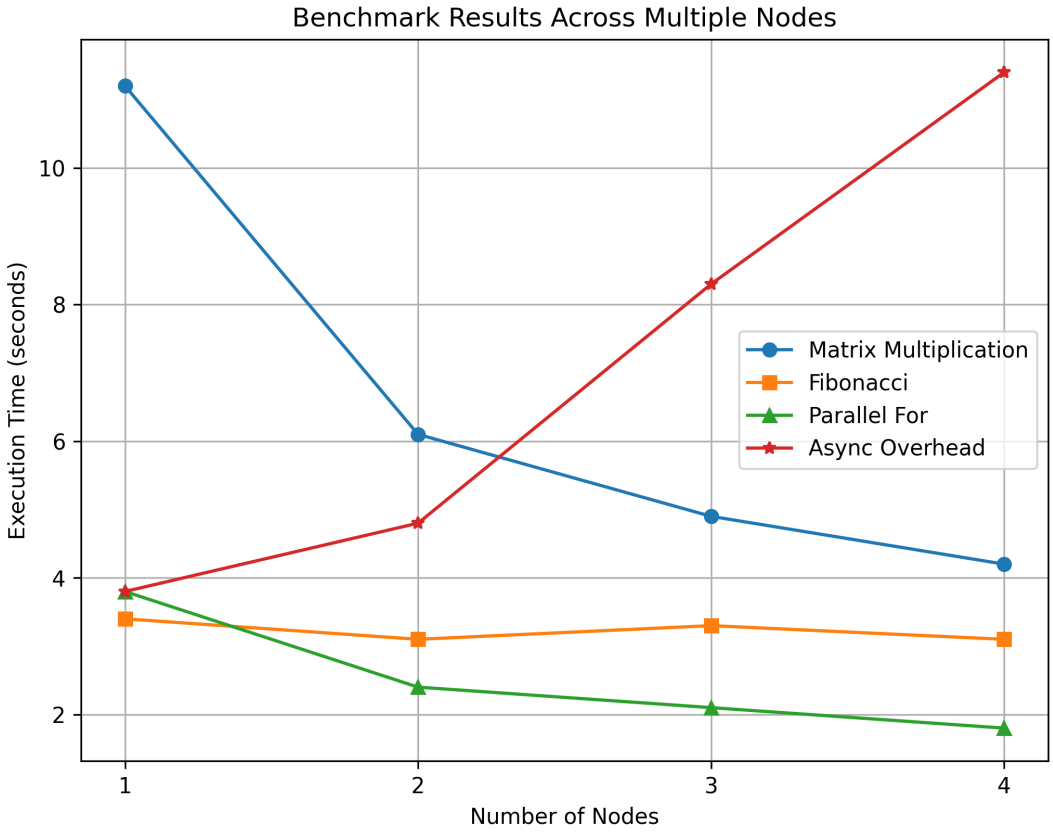

Figure 4.2: Benchmark figures of simple benchmarks for HPX

5. Conclusion

This independent study accomplished its dual goals of bringing MASS C++ up to modern standards and gaining practical familiarity with HPX's programming model. On the MASS C++ side, critical library and security issues were addressed, including upgrading libssh2 for OpenSSL 3 compatibility, removing insecure password handling, enabling ssh-agent support, and improving hostname verification. These changes collectively ensure that the framework can be built and deployed reliably in current environments, and that its remote execution model aligns better with contemporary security practices. The updated sample programs and successful tests across multiple nodes provide a solid baseline for future extensions.

On the HPX side, installation, configuration, and benchmark execution provided concrete insights into a modern asynchronous many-task runtime. Benchmarks such as Fibonacci, matrix multiplication, parallel for loops, and async overhead highlighted both the strengths and limitations of HPX's approach. The results demonstrated strong scalability for data-parallel patterns (matrix multiplication, parallel for) and clarified the costs associated with asynchronous task management. More importantly, these experiments illustrated how HPX exposes distributed parallelism in a way that is expressive and programmable, offering lessons that can inform MASS C++ enhancements.

Together, these efforts position MASS C++ for the next stage of research and development. The work confirmed that modernization is feasible without breaking core abstractions, while HPX provided examples of advanced features futures/promises, actions, modular runtime design that could be selectively adapted. The immediate next steps are to prototype asynchronous APIs within MASS C++, explore hybrid MPI-threading modes, and evaluate more sophisticated serialization and transport layers. With these directions, the thesis can move beyond compatibility fixes to designing new abstractions that improve programmability, performance, and portability for distributed agent-based and data-parallel models.

6. Appendix

MASS C++ Pull Request

https://bitbucket.org/mass_library_developers/mass_cpp_core/pull-requests/4

HPX Benchmarks

/home/NETID/dslab/hpx_benchmarks