Since the advent of parallel algorithms in the C++17 Standard Template Library (STL), the STL has become a viable framework for creating performance-portable applications. Given multiple existing implementations of the parallel algorithms, a systematic, quantitative performance comparison is essential for choosing the appropriate implementation for a particular hardware configuration.

In this work, we introduce a specialized set of micro-benchmarks to assess the scalability of the parallel algorithms in the STL. By selecting different backends, our micro-benchmarks can be used on multi-core systems and GPUs. Using the suite, in a case study on AMD and Intel CPUs and NVIDIA GPUs, we were able to identify substantial performance disparities among different implementations, including GCC+TBB, GCC+HPX, Intel’s compiler with TBB, or NVIDIA’s compiler with OpenMP and CUDA.

Performance Portability, C++, Standard Template Library, Threading Building Blocks, OpenMP, CUDA

1 INTRODUCTION

Writing efficient, parallel applications is notoriously hard, but writing performance-portable, efficient, parallel applications is harder. In the last decades, several types of parallel architectures (like GPUs or Xeon Phis) were introduced, which often required a complete rewrite of the parallelization approach to make applications efficient. To overcome this problem, we have to combine several paradigms such as CUDA [16], OpenMP [6], or MPI [5] to write efficient programs, several frameworks, mainly using C++, such as Kokkos [23] or Raja [2], were proposed to allow scientists to write performance-portable applications. These frameworks allow for an efficient execution of parallel applications using different hardware architectures, i.e., the same program can run on one or more GPUs as well as on multi-core CPUs.

With the advent of C++17, parallel versions of the C++ Standard Template Library (STL) were standardized, which allows ISO C++ parallel programs to be performance portable [15]. Several works have already compared the resulting performance of various performance-portability layers [1, 10]. However, their focus lay on comparing full applications or mini-apps, where significant parts of a rewritten program may significantly influence the resulting performance.

In this work, we set out to devise a set of micro-benchmarks to assess the performance of the individual parallel STL algorithms found in C++ in a quantitative manner. Since different compiler frameworks provide competing implementations of the STL, our goal is to capture the current state-of-the-art of the performance of parallel STL implementations. We compare several combinations of compiler frameworks, including GCC, Intel OneAPI compiler, and NVIDIA HPC SDK, and backends like Intel’s Threading Building Blocks (TBB), High-Performance ParalleX (HPX), and OpenMP.

In particular, we make the following contributions:

1) We introduce the benchmark suite pSTL-Bench, which is an extensible set of micro-benchmarks to assess the performance of parallel STL algorithms on different parallel architectures (multi-cores, GPUs).

2) Using the suite, we conduct a study over a selection of algorithms comparing the performance achieved on current multi-core architectures by different compiler frameworks and backends implementing the parallel STL.

The remainder of the paper is structured as follows. In Section 2, we give an overview of the field by summarizing the related work and current state of the art. Section 3 introduces the specifics of our proposed set of micro-benchmarks. In Section 4 we detail how the experiments were carried out before we show and analyze the experimental results in Section 5. Finally, we draw conclusions from the findings in Section 6 and outline future work.

2 RELATED WORK

Allowing for performance portability has always been a goal for programmers. This is especially true for developers on HPC systems, as novel HPC systems often provide new hardware architectures for which no efficient software solutions exist yet (cf. Jack Dongarra’s interview when receiving the ACM A.M. Turing Award [11]). The Message Passing Interface (MPI) is one of the standards that enables scientists to write efficient, parallel programs that are also portable across architectures. However, MPI has its limits, especially when it comes to efficiently programming multi-core systems or accelerators, such as GPUs or Xeon Phis.

For the reasons mentioned above, several parallel programming frameworks striving for performance portability were introduced. Note that the term “performance portability” may have completely different notions or definitions [18, 19]. Contrary to Pennycook et al.’s restrictive definition, in this work, we consider a program to be performance-portable if it can be executed efficiently on different architectures. We call a program efficient if it performs within a threshold of a specialized program, i.e., a scan algorithm implemented using a performance portability framework performs equally well on an NVIDIA GPU as the best-known CUDA implementation.
Our work focuses on micro-benchmarking algorithms from the ISO C++ standard library. There are several other feature-rich C++ frameworks that offer performance portability. For instance, Kokkos [23] and Raja [2] are one of the most prominent libraries that can be used to program portable HPC applications using different backends, such as SYCL [21], HPX [13], or OpenMP. Although Kokkos may use SYCL as a backend, SYCL itself is a C++ programming model that offers portability between heterogeneous compute resources. Similar to Kokkos, SYCL is a C++ abstraction layer that supports a range of processor architectures and accelerators [10]. For GPUs, the Thrust library [3] provides a high-level interface for STL to C++ programmers. Thrust is used as a backend in the C++ STL implementation of NVIDIA’s High-Performance Computing (HPC) Software Development Kit.

HPX [13] is another C++ abstraction layer for developing efficient parallel and concurrent applications. HPX can be used to write distributed applications using an Active Global Address Space (AGAS), which is an extension to PGAS for transparently moving global objects in between compute nodes [13]. SYCL, HPX, and Kokkos can also be used together to combine their strengths [7].

The benchmark suite SYCL-Bench [14] is a collection of different programs to analyze the performance of various SYCL implementations. Besides micro-benchmarks, SYCL-Bench also features real-world application benchmarks, such as 2DConvolution, as shown in Listings 1 and 2. These lambda functions are called repeatedly to measure the time taken for each operation. The full list of algorithms supported by pSTL-Bench can be found in Table 4 in Appendix A.

3 MICRO-BENCHMARK SUITE PSTL-BENCH

We now present the micro-benchmark suite pSTL-Bench.

3.1 Idea and Goals

As mentioned previously, our primary goal for developing pSTL-Bench is to assess the scalability and the efficiency of the different implementations of the parallel C++ STL. Specifically, we aim to achieve several objectives. First, we would like to determine the sweet-spot of problem size for each parallel STL algorithm, i.e., how large a problem has to be such that utilizing the parallel version is advantageous. Second, we aim to ascertain the maximum number of cores that can be effectively utilized by various parallel algorithms, many of which are memory-bound. Finally, we would like to examine the run-time achievable with each of the parallel STL implementations and backends. We intend to assess the run-time performance achievable with each of the parallel STL implementations and backends. To facilitate this comparison of performance across different standard library implementations, we have compiled these benchmarks into a suite named pSTL-Bench.¹

3.2 Description of Benchmark Kernels

For the sake of conciseness, we select and evaluate five algorithms of different computational structure in this paper. The `find` algorithm performs a linear search on the input array, while the `for_each` call performs a map operation on each element of the input array in parallel. The `reduce` call represents a parallel reduction operation, needed for map-reduce type programming. The `inclusive_scan` represents a typical, parallel prefix-sum operation, and `sort` was selected to include a parallel sorting function into the set of test algorithms.

A precise description of each of the algorithms is shown below:

- `find`: Given an array of \( n \) elements, \( v = [1, 2, \ldots, n] \), find a random element \( v_i \), such that \( v_i \in v \).
- `for_each`: Given an array of \( n \) elements, \( v = [1, 2, \ldots, n] \), compute for each element \( v_j \) such that \( v_j \in v \).
- `inclusive_scan`: Given an array of \( n \) elements, \( v = [1, 2, \ldots, n] \), compute the result of inclusive sum, \( r \), such that \( r_j = \sum_{i=1}^{j} v_i \).
- `reduce`: Given an array of \( n \) elements, \( v = [1, 2, \ldots, n] \), compute \( \sum_{i=1}^{n} v_i \).
- `sort`: Sort the elements of \( v \), where \( v \) is an array of \( n \) elements randomly shuffled, such that \( v_i \in [1, n] \) and \( v_i \neq v_j, \forall j \neq i \).

The full list of algorithms supported by pSTL-Bench can be found in Table 4 in Appendix A.

Since each implementation might have a slightly different interface, the calls to the algorithms are wrapped within lambda functions, as shown in Listings 1 and 2. These lambda functions are then called from a helper function that executes them repeatedly to measure their execution time with Google’s Benchmark library [9], see Listing 3. The wrapping function simply measures the time taken for the invocation of the function to be executed (function \( f \)).

This structure facilitates the incorporation of new benchmarks while the compiler can still inline the functions to avoid the potential overhead of calling the lambdas.

It should be mentioned that it is possible to manually select the input size and data types (\( \text{int}, \text{float}, \text{double}, \text{etc.} \)) used in the benchmarks, therefore, extending its customization and analysis capabilities.

¹Code can be provided on request.
4 EXPERIMENTAL EVALUATION

For acquiring relevant results, experiments are carried out using different configurations in terms of hardware, software, number of threads, input sizes, and memory allocation.

4.1 Hardware and Software Setup

We conduct experiments on three different multi-core, shared-memory systems that comprise 32, 64, and 128 cores, which are called Hydra, Nebula, and VSC-5, respectively. On these systems, we evaluate three different compilers: GCC, Intel oneAPI compiler, and NVIDIA HPC SDK. Also, several backends are tested, which are: Intel’s Threading Building Blocks (TBB), High-Performance ParalleX (HPX), and OpenMP through the implementations of GNU and NVIDIA.

Furthermore, we also perform experiments on two GPU-based systems, Tesla and Ampere, using the NVIDIA HPC SDK with the CUDA compiler.

We provide an overview of the hardware and software details in Table 1.

4.2 Experimental Setup

In our experiments we test the parallel algorithms with different numbers of threads and problem sizes. Specifically, the thread counts range from 1 to the maximum available core count on each machine, so the values 1, 2, 4, . . . , #cores are used. Concerning problem sizes, inputs vary from $2^5$ to $2^{30}$ elements, using 32-bit integers, enabling the testing of a broad set of inputs that are accommodated in various cache levels or necessitate consistent DRAM traffic.

4.3 The Impact of Memory Allocation

Memory allocation might have a significant impact on performance in modern computer architectures. For that reason, a custom parallel allocator is used for the experiments. Through the first-touch policy of Linux, this allocator makes each thread touch the first byte of each object with the given parallel policy, as shown in Listing 4. This enforces the correct placement of threads and memory pages. The parallel allocator used in this work is an adapted version of the NUMA allocator that is part of HPX.

Fig. 1 shows the speedup achieved when using the custom parallel allocator in Hydra. Using this allocator leads to an increase in performance in three of the five algorithms. For the X::find algorithm, the improvement goes from 11% to 39%, from −13% to 74% for the X::inclusive_scan and 56% to 194% for X::reduce. In the X::for_each and the X::sort algorithms, the differences are smaller, always bellow 2.5% and typically around 0.5%.

In only one experiment (X::inclusive_scan), the parallel allocator notably reduced performance, that is with the NVIDIA compiler and the OpenMP backend.

Considering the results of the parallel allocator, the rest of the benchmarks are executed using it except for those tests related to HPX, which is incompatible with it.

5 EXPERIMENTAL RESULTS

Table 2 summarises the results obtained, showing the speedup compared to the sequential implementation with GCC. In this table, we use a fixed baseline performance, GCC’s sequential performance in this case, which may lead to speedups that are large than the total core count.

Table 3 shows the maximum number of threads for which parallel efficiency is above 70%. The purpose of this table is to analyze the number of threads that can be effectively utilized without excessively wasting resources. The threshold of 70% is somewhat arbitrarily defined, but it assists in estimating the effectiveness of the parallelization.

---

Footnotes:
3The names of the machines were intentionally anonymized for the dual-anonymous review process.

---

Listing 1: Wrapper of the for_each benchmark using STL.

```cpp
1. template <class Policy, class Function>
2. static void benchmark_for_each_wrapper(benchmark::State & state, Function && f) {
3.   auto data = suite::generate_increment<Policy>(
4.       const auto & size = state.range(0);
5.       auto execution_policy = Policy();
6.   }
7.   std::for_each(execution_policy, data.begin(),
8.       data.end(), f);
9. }
```

Listing 2: Wrapper of the for_each benchmark using GNU’s implementation.

```cpp
1. template <class Policy, class Function>
2. static void benchmark_for_each_wrapper(benchmark::State & state, Function && f) {
3.   auto data = suite::generate_increment<Policy>(
4.       const auto & size = state.range(0);
5.       auto execution_policy = Policy();
6.   }
7.   std::for_each(execution_policy, data.begin(),
8.       data.end(), f);
9. }
```

Listing 3: Example of the benchmark wrapper for for_each.

```cpp
1. template <class Policy, class Function>
2. static void benchmark_for_each_wrapper(benchmark::State & state, Function && f) {
3.   auto data = suite::generate_increment<Policy>(
4.       const auto & size = state.range(0);
5.       auto execution_policy = Policy();
6.   }
7.   std::for_each(execution_policy, data.begin(),
8.       data.end(), f);
9. } // return the overall memory block
10. }
```

Listing 4: Function allocate for the custom parallel allocator.

```cpp
1. template <size_type cnt, void const * = nullptr>
2. static void allocate(size_type cnt, void const * = nullptr) {
3.   // allocate memory
4.   auto p = static_cast<pointer>(::operator new(cnt * sizeof(T)));
5.   // touch the first byte of every object
6.   auto & input_data = suite::generate_increment<Policy>(
7.       const auto & size = state.range(0);
8.   auto data = suite::generate_increment<Policy>(
9.       execution_policy, size, 1);
10.   for (auto _ : state) {
11.     WRAP_TIMING(f(execution_policy, data, kernel));
12.   }
13.   state_.SetBytesProcessed(state_.iterations() * data.
14.       size() * sizeof(int));
15. }
```

---

---
Table 1: Summary of the hardware and software used in our study.

<table>
<thead>
<tr>
<th>machine</th>
<th>Hydra</th>
<th>Nebula</th>
<th>VSC-5</th>
<th>Tesla</th>
<th>Ampere</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU/GPU</td>
<td>Intel Xeon 6130F</td>
<td>AMD EPYC 7551</td>
<td>AMD EPYC 7713</td>
<td>NVIDIA Tesla P4</td>
<td>NVIDIA Ampere A2</td>
</tr>
<tr>
<td>core frequency</td>
<td>2.10 GHz</td>
<td>2.00 GHz</td>
<td>2.00 GHz</td>
<td>1.11 GHz</td>
<td>1.77 GHz</td>
</tr>
<tr>
<td>#sockets</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>#cores (node)</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>2560</td>
<td>1280</td>
</tr>
<tr>
<td>#threads (node)</td>
<td>32</td>
<td>64</td>
<td>256</td>
<td>2560</td>
<td>1280</td>
</tr>
<tr>
<td>compilers</td>
<td>g++ 12.1.0</td>
<td>g++ 12.3.0</td>
<td>g++ 12.2.0</td>
<td>g++ 10.2.1</td>
<td>g++ 10.2.1</td>
</tr>
<tr>
<td>icpx 2021.7.0</td>
<td>icpx 2021.7.0</td>
<td>icpx 2021.7.0</td>
<td>icpx 2021.7.0</td>
<td>icpx 2021.7.0</td>
<td>icpx 2021.7.0</td>
</tr>
<tr>
<td>libraries</td>
<td>TBB 2021.9.0</td>
<td>TBB 2021.7.0</td>
<td>TBB 2021.9.0</td>
<td>CUDA 11.8</td>
<td>CUDA 12.2</td>
</tr>
<tr>
<td>GOMP (from g++ 12.1.0)</td>
<td>GOMP (from g++ 12.3.0)</td>
<td>GOMP (from g++ 12.2.0)</td>
<td>GOMP (from g++ 12.2.0)</td>
<td>GOMP (from g++ 12.2.0)</td>
<td>GOMP (from g++ 12.2.0)</td>
</tr>
<tr>
<td>NVOMP 22.11</td>
<td>NVOMP 22.11</td>
<td>NVOMP 22.11</td>
<td>NVOMP 22.11</td>
<td>NVOMP 22.11</td>
<td>NVOMP 22.11</td>
</tr>
</tbody>
</table>

Figure 1: Speedup when using custom parallel allocator with 32 threads and a problem size of $2^{30}$ elements in Hydra. Higher is better.

All the data presented in this section are derived from the median of ten executions.

5.1 X::find

Figure 2 shows how the execution times grow with the size of the problem in Hydra. The behavior for other machines is shown in Appendix B. For small problem sizes, the sequential implementation offers better execution times, with a difference of orders of magnitude, highlighting the overhead of launching parallel sections. This difference is consistently reduced until problem sizes of around $2^{16}$, where the performance of the parallelism starts to compensate for the overhead. It is interesting to note that the GNU parallel implementation falls back to the sequential implementation until the problem size reaches $2^9$. From problem sizes of $2^{20}$, parallel implementations begin to be significantly faster than the sequential one. These behaviors, regarding the input size, can be extended to the rest of the benchmarks.

Related to the speedup, efficiency is an issue, where the maximum speedup achieved was $10 \times$ with the Intel compiler and using TBB with 32 threads in Hydra. Also, using more than 32 threads results in little, if any, gains in performance.

5.2 X::for_each

Figure 3 shows the speedup obtained for a problem size of $2^{30}$ elements. Note that the Intel compiler and, particularly, the NVIDIA compiler achieve a better performance than GCC. Profiling of the code indicates that besides using its math library, the NVIDIA and Intel compilers make better use of vector instructions in comparison to GNU’s. Interestingly, the NVIDIA compiler uses 256-bit instructions, while the Intel compiler uses only 128-bit instructions.

Additionally, data in Table 3 shows an almost perfect scaling. This is reasonable given the embarrassingly parallel nature of the algorithm.

5.3 X::inclusive_scan

First, it should be noted that the GNU’s collection of parallel algorithms does not include the inclusive_scan function.
Table 2: Speedup against GCC’s sequential implementation for machines Hydra, Nebula, and VSC-5, with 32, 64, and 128 cores, respectively. Notation is Hydra/Nebula/VSC-5. Higher is better.

<table>
<thead>
<tr>
<th></th>
<th>X::find</th>
<th>X::for_each</th>
<th>X::inclusive_scan</th>
<th>X::reduce</th>
<th>X::sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC-TBB</td>
<td>9.54 / 3.99 / 5.83</td>
<td>31.92 / 54.31 / 85.18</td>
<td>4.72 / 3.31 / 4.80</td>
<td>12.29 / 6.25 / 7.14</td>
<td>9.51 / 10.04 / 9.42</td>
</tr>
<tr>
<td>GCC-GNU</td>
<td>6.72 / 1.46 / 2.65</td>
<td>31.68 / 54.15 / 85.01</td>
<td>– / – / –</td>
<td>15.08 / 5.49 / 7.20</td>
<td>28.09 / 28.07 / 65.36</td>
</tr>
<tr>
<td>GCC-HPX</td>
<td>7.30 / 1.07 / 2.31</td>
<td>29.17 / 50.52 / 64.33</td>
<td>2.18 / 0.79 / 0.84</td>
<td>9.46 / 1.14 / 1.76</td>
<td>10.24 / 9.10 / 8.46</td>
</tr>
<tr>
<td>NVC-OMP</td>
<td>4.78 / 1.07 / 1.31</td>
<td>224.55 / 305.65 / 608.00</td>
<td>0.88 / 0.62 / 0.96</td>
<td>23.77 / 17.91 / 11.54</td>
<td>9.14 / 7.45 / 6.76</td>
</tr>
</tbody>
</table>

Table 3: Maximum number of threads for which parallel efficiency is above 70 % (compared to the execution with 1 thread) for machines Hydra, Nebula, and VSC-5. Notation is Hydra/Nebula/VSC-5. Higher is better.

<table>
<thead>
<tr>
<th></th>
<th>X::find</th>
<th>X::for_each</th>
<th>X::inclusive_scan</th>
<th>X::reduce</th>
<th>X::sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC-TBB</td>
<td>1 / 2 / 1</td>
<td>32 / 64 / 32</td>
<td>1 / 1 / 1</td>
<td>8 / 4 / 1</td>
<td>4 / 8 / 4</td>
</tr>
<tr>
<td>GCC-GNU</td>
<td>1 / 1 / 1</td>
<td>32 / 64 / 32</td>
<td>– / – / –</td>
<td>1 / 1 / 1</td>
<td>32 / 32 / 16</td>
</tr>
<tr>
<td>GCC-HPX</td>
<td>16 / 4 / 1</td>
<td>32 / 64 / 8</td>
<td>4 / 2 / 1</td>
<td>8 / 2 / 1</td>
<td>8 / 4 / 4</td>
</tr>
<tr>
<td>NVC-OMP</td>
<td>8 / 4 / 1</td>
<td>32 / 32 / 128</td>
<td>1 / 1 / 1</td>
<td>1 / 2 / 1</td>
<td>2 / 2 / 4</td>
</tr>
</tbody>
</table>

Figure 3: Speedup against GCC’s sequential implementation for benchmark X::for_each. Higher is better.

Based on the results shown in Figures 4, 5, and Table 2, it is evident that the NVIDIA implementation yields limited speedup for this operation, being even slower than the sequential version. This is because this implementation falls back to a sequential execution, which is negatively affected by the custom parallel allocator used.

The rest of the implementations produce slight improvements in performance when using parallelism. For example, a maximum speedup of 4.81× can be achieved using TBB as the backend. However, the efficiency of these implementations is far from ideal.

5.4 X::reduce

It should be noted that the GNU’s collection of algorithms does not include a reduce function so the accumulate function has been used instead.

As shown in Figure 6 the speedup behavior changes from system to system. In Hydra (Figure 6a), for example, a good speedup is achieved up to 16 threads, with a maximum speedup of 23.77×.

However, the performance achieved with the NVIDIA compiler with the OpenMP backend stands out, especially in Nebula and VSC-5. In these systems, the rest of the compilers and backends present poor scalability, especially when more than 16 threads are used, see Figures 6b and 6c.

5.5 X::sort

Two different factors are relevant when analyzing the results for this benchmark, the problem size and the number of threads.

For small sizes (see Figure 7), below 212, the NVC-OMP shows a significantly higher overhead than the rest of compilers and backends. For bigger sizes, it is the GNU implementation that obtains better results.

Regarding the number of threads, when this is 8 or less, the NVIDIA compiler achieves a considerably better performance than the rest, see Figure 8. Nevertheless, with more threads, the implementation of GNU which uses OpenMP establishes itself as a clear winner.

It is interesting to note that GNU’s efficiency is above 70 % up to 32 threads in Hydra and Nebula, but only up to 16 threads in...
Ruben Laso, Diego Krupitza, and Sascha Hunold

Figure 4: Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Benchmark `X::inclusive_scan`. Lower is better.

Figure 5: Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Benchmark `X::inclusive_scan`. Higher is better.

Figure 6: Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Benchmark `X::reduce`. Higher is better.

VSC-5, see Table 3. The rest of the backends, cannot use more than 8 threads while keeping an efficiency of 70% or better.

5.6 Performance on GPUs

Figure 9 includes the execution times of the benchmarks on Hydra, Tesla and Ampere. Results show that both the input size and the particular algorithm play a key role in terms of performance for the GPUs compared to the CPU.

First, a substantial penalty is paid in the form of overhead when using a small input size. Also, transferring a large amount of data seems to penalize the performance of the GPUs as well, see Figures 9c and 9d.

Regarding the algorithms, the `X::find` and `X::reduce` have poor performance on the GPUs. However, the `X::for_each`, the `X::inclusive_scan`, and the `X::sort` achieve significantly better execution times, especially for the sizes ranging from $2^{16}$ to $2^{27}$ elements.
6 CONCLUSIONS

In this work, we introduced a specialized set of benchmarks named pSTL-Bench to assess the performance of the parallel algorithms present in the C++ standard template library. With this suite, we aim to facilitate the users comparing compilers and backends to decide which fits best the particular characteristics of their system.

Additionally, using this set of benchmarks, we conducted a comprehensive analysis of performance on current multi-core architectures by benchmarking some of the most used algorithms in STL. This analysis took into account not only the compiler and backend using but also the input size, the number of threads used, and the memory allocation.

First, we showed the potential performance improvements when using a custom parallel allocator in NUMA systems. Generally, execution times are improved, up to 194% in the best case, and minor losses in some scenarios.

Second, when using small input sizes it is possible to note the large overhead of launching parallel sections and the differences between backends, being the TBB typically lower than those based on OpenMP or HPX. It is interesting to note as well that the GNU implementation falls back to the sequential execution for inputs lower than 512 elements.

On the other hand, for large input sizes, the parallel implementations show better in the vast majority of scenarios. For example, the speedup and efficiency in the `X::for_each` stand out compared to the rest of the algorithms. Nevertheless, some algorithms like `X::find` or `X::inclusive_scan` show poor speedup and, consequently, a bad efficiency.

Third, it is interesting to note that there might be large differences in performance due to vectorization. In our experiments, we found that the NVIDIA compiler applies vector instruction more often than the rest of the compilers, particularly in the `X::for_each` algorithm.

Finally, we evaluated the performance of the CUDA backend for the NVIDIA compiler with two different GPUs. We found high penalties for small inputs, but also for high inputs where the data transfers dominate the execution times. Nevertheless, for medium problem sizes and some particular algorithms, the GPUs are capable of outperforming the CPUs.

Regarding future work, we would like to expand our benchmark suite, to include more algorithms and also to support more compilers and backends. Similarly, it would be interesting to extend our experimental analysis on more architectures, particularly GPUs and, potentially, FPGAs.
Figure 9: Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

REFERENCES


## A SUPPORT OF PARALLEL IMPLEMENTATIONS

Table 4: List of algorithms with support for parallelization in STL. Those with benchmarks in pSTL-Bench are noted in gray.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Algorithm</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacent_difference</td>
<td>adjacent_find</td>
<td>all_of</td>
</tr>
<tr>
<td>any_of</td>
<td>copy</td>
<td>copy_if</td>
</tr>
<tr>
<td>copy_n</td>
<td>count</td>
<td>count_if</td>
</tr>
<tr>
<td>destroy</td>
<td>destroy_n</td>
<td>equal</td>
</tr>
<tr>
<td>exclusive_scan</td>
<td>fill</td>
<td>fill_n</td>
</tr>
<tr>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_if_not</td>
<td>for_each</td>
</tr>
<tr>
<td>for_each_n</td>
<td>generate</td>
<td>generate_n</td>
</tr>
<tr>
<td>includes</td>
<td>inclusive_scan</td>
<td>inplace_merge</td>
</tr>
<tr>
<td>is_heap</td>
<td>is_heap_until</td>
<td>is_partitioned</td>
</tr>
<tr>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
</tr>
<tr>
<td>mimmax_element</td>
<td>mismatch</td>
<td>move</td>
</tr>
<tr>
<td>none_of</td>
<td>nth_element</td>
<td>partial_sort</td>
</tr>
<tr>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
</tr>
<tr>
<td>remove_copy_if</td>
<td>remove_if</td>
<td>replace</td>
</tr>
<tr>
<td>replace_copy</td>
<td>replace_copy_if</td>
<td>replace_if</td>
</tr>
<tr>
<td>reverse</td>
<td>reverse_copy</td>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
</tr>
<tr>
<td>set_difference</td>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
</tr>
<tr>
<td>set_union</td>
<td>sort</td>
<td>stable_sort</td>
</tr>
<tr>
<td>stable_partition</td>
<td>swap_ranges</td>
<td>transform</td>
</tr>
<tr>
<td>transform_exclusive_scan</td>
<td>transform_inclusive_scan</td>
<td>transform_reduce</td>
</tr>
<tr>
<td>uninitialized_copy</td>
<td>uninitialized_copy_n</td>
<td>uninitialized_default_construct</td>
</tr>
<tr>
<td>uninitialized_default_construct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uninitialized_move</td>
<td>uninitialized_move_n</td>
<td>uninitialized_value_construct</td>
</tr>
<tr>
<td>uninitialized_value_construct</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B \texttt{X::FIND}

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 10: Results for benchmark \texttt{X::find}; Hydra.

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 12: Results for benchmark \texttt{X::find}; VSC-5.
C  X::FOR_EACH

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 13: Results for benchmark X::for_each; Hydra.

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 14: Results for benchmark X::for_each; Nebula.

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 15: Results for benchmark X::for_each; VSC-5.
D X::INCLUSIVE_SCAN

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 16: Results for benchmark X::inclusive_scan; Hydra.

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 17: Results for benchmark X::inclusive_scan; Nebula.

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 18: Results for benchmark X::inclusive_scan; VSC-5.
E  X::REDUCE

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 19: Results for benchmark X::reduce; Hydra.

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 20: Results for benchmark X::reduce; Nebula.

(a) Execution time scalability with the problem size. All cores used except for GCC’s seq. implementation. Lower is better.

(b) Execution time scalability with the number of threads used. Problem size is $2^{30}$. Lower is better.

(c) Speedup against GCC’s seq. implementation. Problem size is $2^{30}$. Higher is better.

Figure 21: Results for benchmark X::reduce; VSC-5.
Figure 22: Results for benchmark \texttt{X::sort}; \textit{Hydra}.

Figure 23: Results for benchmark \texttt{X::sort}; \textit{Nebula}.

Figure 24: Results for benchmark \texttt{X::sort}; \textit{VSC-5}.