Exploring Scalability in C++ Parallel STL Implementations

Ruben Laso
ruben.laso@tuwien.ac.at
Faculty of Informatics
TU Wien
Vienna, Austria

Diego Krupitza
krupitza@par.tuwien.ac.at
Faculty of Informatics
TU Wien
Vienna, Austria

Sascha Hunold
sascha.hunold@tuwien.ac.at
Faculty of Informatics
TU Wien
Vienna, Austria

ABSTRACT

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.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**.

KEYWORDS

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

ACM Reference Format:
Ruben Laso, Diego Krupitza, and Sascha Hunold. 2024. Exploring Scalability in C++ Parallel STL Implementations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 the problem of having to combine several paradigms such as CUDA [18], OpenMP [7], or MPI [6] to write efficient programs, several frameworks, mainly using C++, such as Kokkos [26] 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 [16]. Several works have compared the resulting performance of various performance-portability layers [1, 11]. However, their focus lay on comparing full applications or mini-apps, where specific 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 compilers, including GCC, Intel OneAPI compiler, and NVIDIA HPC SDK, and backends like Intel's Threading Building Blocks (TBB), High-Performance ParalleX (HPX), OpenMP, and CUDA.

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. Our results show that there are significant performance differences between the available backends.

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 [12]). The Message Passing Interface (MPI) is one of the standards that enables scientists to write efficient, parallel programs that are also

Conference'17, July 2017, Washington, DC, USA
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [20, 21]. Contrary to Pennycook et al.'s restrictive definition, in this work, we consider a program to be performance-portable 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 [26] 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 [23], HPX [14], 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 [11]. 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 [14] 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 [14]. SYCL, HPX, and Kokkos can also be used together to combine their strengths [8].

The benchmark suite SYCL-Bench [15] 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 well as so-called runtime benchmarks, such as a DAG benchmark with many independent tasks. Pennycook et al. presented an analysis of the terminology used in SYCL and ISO C++ to shed light on the relationship between C++ concepts in both programming models [19].

This work was inspired by Lin et al. [16], who examined the ISO C++ capabilities for creating performance-portable applications. In their work, they developed an ISO C++ parallel version of the BabelStream benchmark [9] and also adapted real-world applications, such as MiniBude, to utilize the parallel STL of ISO C++.

In the context of the present work, we will evaluate different implementations of the parallel STL for multi-core systems. That collection includes the HPX [14] library discussed above. We also analyze the performance of the GNU's parallel STL, based on MCSTL [24] which uses OpenMP for parallelization and can be seen as the original parallel implementation of the STL. Currently, the GNU and the Intel compilers use the Intel Threading Building Blocks (TBB) [22] for implementing the parallel C++ STL, whereas NVIDIA relies on its Thrust [3] library, which uses OpenMP [7] and CUDA [18] for its parallel implementation of the STL.

Compared to the previous benchmarks, pSTL-Bench targets the scalability of the individual building blocks, in the same spirit as the OMPTB [13] or the EPCC OpenMP microbenchmark suite [5] for benchmarking OpenMP primitives.

Listing 1: Kernel for the `for_each` benchmark.

```
const auto kernel = [](auto & input , const auto k_it) {
  volatile size_t I = k_it;
  pstl::elem_t a{};
  for (auto i = 0; i < I; ++i) { a++; }
  input = a;
};
```

3 MICRO-BENCHMARK SUITE PSTL-BENCH

pSTL-Bench is a micro-benchmark suite designed to assess the scalability and efficiency of the different implementations of the parallel C++ STL. The availability of such a benchmark suite will help us to answer the following research questions. (1) What is the sweet spot in terms 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? This also includes an analysis on which problem and problem size is better suited for running on a GPU instead of a CPU. (2) What is the maximum number of cores that can be effectively utilized by parallel STL algorithms, as many of which are memory-bound? (3) How do the different STL implementations and backends compare to each other in terms of run-time?

To facilitate this study across different standard library implementations, we have compiled these benchmarks into a suite named pSTL-Bench.[1] The list of algorithms supported by pSTL-Bench can be found in Table 1.

3.1 Description of Benchmark Kernels

Due to space constraints, we analyze five algorithms featuring distinct computational patterns: (1) The `find` algorithm performs a linear search on an input array, while (2) the `for_each` call performs a map operation on each element of the input array in parallel. (3) The `reduce` call represents a parallel reduction operation, needed for map-reduce type programming. (4) The `inclusive_scan` represents a typical, parallel prefix-sum operation, and (5) `sort` represents a fundamental function when designing algorithms.

A more detailed, formal description of each of the five selected 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 the result of applying the kernel shown in Listing 1 to each element of v. The computational load of the kernel can be adjusted by changing the number of iterations in the loop, which we will refer to as k_{it} in the remainder of this paper. Note that the number of iterations is stored in a `volatile` variable to prevent the compiler from optimizing the loop.

`inclusive_scan`: Given an array of n elements, $v = [1, 2, \ldots, n]$, compute the result of inclusive sum, r, such that $r_i = \sum_{j=1}^{i} v_j$.

`reduce`: Given the array $v = [1, 2, \ldots, n]$, 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 $v_i \neq v_j, \forall i \neq j$.

[1]Code can be provided on request.

Table 1: Algorithms that allow execution policies in STL. Algorithms supported in pSTL-Bench are colored in gray.

adjacent_difference ▨	adjacent_find ▨	all_of ▨	any_of ▨
copy ▨	copy_if ▨	copy_n ▨	count ▨
count_if ▨	destroy	destroy_n	equal ▨
exclusive_scan ▨	fill ▨	fill_n	find ▨
find_end	find_first_of	find_if	find_if_not
for_each ▨	for_each_n	generate ▨	generate_n ▨
includes ▨	inclusive_scan ▨	inplace_merge ▨	is_heap ▨
is_heap_until	is_partitioned	is_sorted ▨	is_sorted_until ▨
lexicographical_compare ▨	max_element ▨	merge ▨	min_element ▨
minmax_element	mismatch ▨	move	none_of ▨
nth_element	partial_sort ▨	partial_sort_copy	partition ▨
partition_copy	reduce ▨	remove*	replace*
reverse	reverse_copy	rotate	rotate_copy
search ▨	search_n	set_difference ▨	set_intersection ▨
set_symmetric_difference	set_union	sort ▨	stable_sort
stable_partition	swap_ranges	transform ▨	transform_exclusive_scan ▨
transform_inclusive_scan ▨	transform_reduce ▨	uninitialized_*	unique*

Since each implementation might have a slightly different interface, the calls to the algorithms are wrapped within lambda functions, as shown in Listings 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 [10], see Listing 3. The wrapping function measures the time taken for the invocation of the function to be executed (function f).

3.2 Benchmark Features

The **number of threads** to be used by the parallel algorithms can be set by the user through the environment variable OMP_NUM_THREADS. This variable is used by default by the OpenMP-based backends and is also captured to set up the number of threads for the TBB backend. For determining the number of threads to be used by the HPX backend, the user can set the argument --hpx:threads=N, where N is the number of threads to be used.

Listing 3 shows the function SetBytesProcessed() and the macro WRAP_TIMING. The former sets the number of bytes processed by the algorithm, which can be used to **analyze its throughput**. The latter, detailed in Listing 4, is a macro that wraps the execution of the function f. It measures not only the execution time but also enables the support for hardware counters to extract additional performance metrics, such as the number of cache misses or floating-point operations. Currently, the support for **hardware performance counters (HPCs)** is available through the high-level API of PAPI [4] and the Marker API of Likwid [25]. By utilizing this feature, pSTL-Bench ensures that HPCs are only captured for the actual STL call, excluding both the environment setup and the initialization of data structures. Other tools like perf could be used to obtain HPCs for the whole program, but other program parts (e.g., shuffling the data before X::sort) could introduce noise into the measurements.

Finally, it is possible to **change** the predefined **input sizes and data types** (int, float, double, etc.) used in the benchmarks. The benchmark suite can therefore easily be extended and adjusted to specific performance requirements.

Listing 2: Wrappers for the sort benchmark with the std (top) and the GNU backend (bottom).

```cpp
auto sort_std = [](auto && policy, auto & input) {
  std::sort(policy, input.begin(), input.end());
};
auto sort_gnu = [](auto && policy, auto & input) {
  __gnu_parallel::sort(input.begin(), input.end());
};
```

Listing 3: Example of the benchmark wrapper for sort.

```cpp
template<class Policy, class Function>
void wrapper(benchmark::State & state, Function && f) {
  const auto & size = state.range(0);
  auto data = pstl::generate_increment(Policy{}, size);

  std::random_device rd;
  std::mt19937 g(rd());

  for (auto _ : state) {
    std::shuffle(data.begin(), data.end(), g);
    WRAP_TIMING(f(Policy{}, data);)
  }

  state.SetBytesProcessed(state.iterations() * data.size
      () * sizeof(pstl::elem_t));
}
```

Listing 4: Wrapper WRAP_TIMING for measuring the execution time of a benchmark.

```cpp
#define WRAP_TIMING(code)                              \
        pstl::hw_counters_begin(state);                \
        MEASURE_TIME(code);                            \
        pstl::hw_counters_end(state);                  \
        state.SetIterationTime(_seconds.count());
```

3.3 Memory allocation

Memory allocation can have a significant impact on performance in modern, parallel computer architectures, in particular in NUMA architectures. For that reason, pSTL-Bench uses a custom parallel

Listing 5: Function `allocate` for the custom parallel allocator.

```
pointer allocate(size_type cnt, void const * = nullptr) {
  auto p = static_cast<pointer>(::operator new(cnt *
      sizeof(T)));

  // touch the first byte of every object
  std::for_each(execution_policy, begin, begin + cnt,
    [](T & val) { *reinterpret_cast<char*>(&val) = 0; });

  return p;
}
```

allocator to implement a first-touch policy. By relying on this allocator, each thread touches the first byte of each object with the given parallel policy, as shown in Listing 5. This strategy enforces the correct placement of threads and memory pages, which is important for efficiently utilizing the available DRAM bandwidth of all NUMA nodes on a multi-core system. Our parallel allocator is an adapted version of the NUMA allocator that is part of HPX.[2]

4 EXPERIMENTAL SETUP

The 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 multi-core, shared-memory systems that comprise 32, 64, and 128 cores, which are called *Mach A*, *Mach B*, and *Mach C*, respectively.[3] For the sake of clarity, we refer to these systems also as *Mach A* (Skylake), *Mach B* (Zen 1), and *Mach C* (Zen 3), highlighting the underlying architecture.

On the multi-core machines, we evaluate three different compilers: GCC, Intel oneAPI compiler, and NVIDIA HPC SDK. We also tested the following backends: Intel's Threading Building Blocks (TBB), High-Performance ParalleX (HPX), and OpenMP through the implementations of GNU and NVIDIA.

Furthermore, we perform experiments on two GPU-based systems, *Mach D* (Tesla) and *Mach E* (Ampere), using the NVIDIA HPC SDK with the CUDA compiler.

We intentionally avoid using 256 threads on *Mach C* (Zen 3) to prevent the use of multi-threading, which could lead to performance degradation and is typically disabled in HPC environments. On *Mach A* (Skylake) and *Mach B* (Zen 1), multi-threading is disabled, and the number of threads is set to the number of physical cores.

Table 2 provides a summarized overview of the hardware and software.

4.2 Experimental Details

We begin by providing experimental details to facilitate understanding and potential reproduction of the results presented in the following sections.

We test the parallel algorithms with different thread counts and problem sizes. Specifically, thread counts range from 1 to the maximum available core count on each machine, and thus, the values $1, 2, 4, \ldots,$ #cores are used. Problem sizes vary from 2^3 to 2^{30} elements, using 64-bit floating-point operands, enabling the testing of a broad set of inputs that are accommodated in various cache levels or necessitate consistent DRAM traffic. We avoid any pinning or providing hints to the runtime systems about the placement of threads and data for two reasons: first, to prevent introducing any bias in the results, and second, to evaluate the runtimes' capabilities to determine the best placement of threads and data.

When allocating the jobs in task managers, such as Slurm [27], the exclusive use of the node is requested to avoid interference from other users. Then, the number of threads is set accordingly using the environment variables and options detailed in Section 3.2.

We present speedup charts (for example, Fig. 3) using a log-linear scale. Therefore, the x-axis is logarithmic, representing the number of threads, and the y-axis is linear, representing the speedup. Consequently, the ideal speedup is represented by a curved line. This is a conscious decision to allow for a better comparison of the results. When using linear-linear plots, it is difficult to observe effects for a small number of threads, as most values are on the left side of the plot. On the other hand, using log-log plots might lead to a misinterpretation of the results, where low speedups seem closer to the ideal speedup than they actually are.

For further insights into the performance of the parallel algorithms, we use Likwid to gather statistics from the hardware performance counters, such as the number of instructions executed, use of vectorization, and memory bandwidth. These statistics are extracted via the Likwid Marker API, as explained in Section 3.2.

All the data presented in Section 5 are derived from the average running time provided by Google Benchmark when using the option `--benchmark_min_time=5s`, i.e., each test is executed for at least 5 s (up to 10^9 iterations).

5 EXPERIMENTAL RESULTS

This section gathers the results of the experiments conducted to evaluate the performance of the parallel allocator, the algorithms, the backends, and the GPU offloading.

5.1 The Impact of Memory Allocation

The first set of experiments is designed to evaluate the impact of pSTL-Bench's allocator (see Section 3.3) on the performance of the parallel algorithms.

Figure 1 shows the speedup achieved using the custom parallel allocator compared to the default allocator on *Mach A* (Skylake). There are two scenarios where the custom allocator has a significant impact on the performance: the `X::for_each` with small operational intensity and the `X::reduce`. In these cases, the speedup is up to 63 % and 50 %, respectively. There are some cases where the custom allocator makes no significant difference, most notably the `X::for_each` when the computational intensity is high ($k_{\text{it}} = 1000$) and the `X::sort` regardless of the compiler. Nevertheless, for the `X::find` and `X::inclusive_scan` algorithms, the custom allocator affects the performance negatively, with a decrease of up to 24 % and 19 %, respectively.

[2]For the NUMA allocator of HPX, see https://github.com/STEllAR-GROUP/hpx/blob/fe048ee6e01abedad0a60a0fdc204116419871c3/libs/core/compute_local/include/hpx/compute_local/host/numa_allocator.hpp

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

Table 2: Summary of the hardware and software used in our study.

Machine	*Mach A* (Skylake)	*Mach B* (Zen 1)	*Mach C* (Zen 3)	*Mach D* (Tesla)	*Mach E* (Ampere)
CPU/GPU	Intel Xeon 6130F	AMD EPYC 7551	AMD EPYC 7713	NVIDIA Tesla T4	NVIDIA Ampere A2
CPU/GPU arch.	Skylake	Zen	Zen 3	Turing	Ampere
Core frequency	2.10 GHz	2.00 GHz	2.00 GHz	1.11 GHz	1.77 GHz
Sockets | NUMA nodes	2 | 2	2 | 8	2 | 8	1 | 1	1 | 1
Total #cores | threads	32 | 32	64 | 64	128 | 256	2560 | 2560	1280 | 1280
Max. #threads used	32	64	128	2560	1280
Memory (node / GPU)	48 GiB	32 GiB	512 GiB	16 GiB	8 GiB
Memory (per core)	1.5 GiB	0.5 GiB	4 GiB	—	—
Compilers	g++ 12.1.0	g++ 12.3.0	g++ 12.2.0	g++ 10.2.1	g++ 10.2.1
	nvc++ 22.11	nvc++ 23.7	nvc++ 22.9	nvc++ 23.5	nvc++ 23.5
	icpx 2021.7.0		icpx 2022.2.1		
Libraries	TBB 2021.9.0	TBB 2021.10.0	TBB 2021.7.0	CUDA 11.8	CUDA 12.2
	HPX 1.9.0	HPX 1.9.1	HPX 1.8.1		
	GOMP (g++ 12.1.0)	GOMP (g++ 12.3.0)	GOMP (g++ 12.2.0)		
	NVOMP 22.11	NVOMP 22.11	NVOMP 22.9		
STREAM BW 1 | all core(s) (GB/s)	11.7 | 135	26.0 | 204	42.6 | 249	N/A | 264	N/A | 172

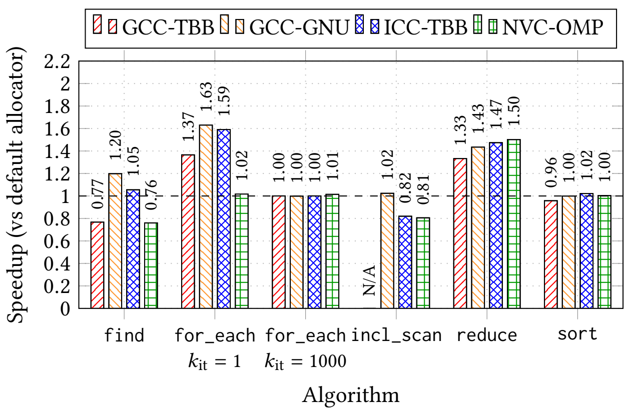


Figure 1: Speedup when using custom parallel allocator with 32 threads and a problem size of 2^{30} elements on *Mach A* (Skylake). Higher is better.

The GCC-GNU backend benefits the most, improving or maintaining the performance in all cases. Conversely, the NVC-OMP backend is the most sensitive to the custom allocator, varying from a −24 % to a +50 % change in performance.

Based on the results obtained with the parallel allocator, we use it in the rest of the experiments since the benefits of using the custom allocator outweigh the drawbacks. The only exceptions are the HPX backend, which has its own memory allocation strategy, and the CUDA backend, which uses the GPU's memory.

5.2 `X::for_each`

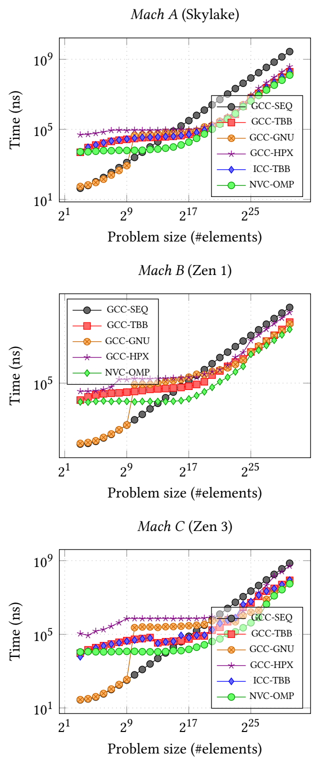
The first benchmark we analyze is `X::for_each`, which is a simple loop that applies a function to each element of a container. This benchmark is interesting because it can provide a good insight into the overhead of the different parallel backends, especially when the arithmetic intensity is low. For higher arithmetic intensity, the performance is expected to be closer to the ideal speedup since actual computation is the dominant factor.

Figure 2 shows the execution times for the `X::for_each` benchmark with the minimal arithmetic intensity ($k_{it} = 1$). Results are consistent across the three machines, with the sequential implementation outperforming the parallel implementations for small problem sizes, up to 2^{10} elements. For larger problem sizes, the parallel implementations start to compensate for the overhead, ultimately being faster than the sequential implementation for problem sizes of around 2^{16} elements.

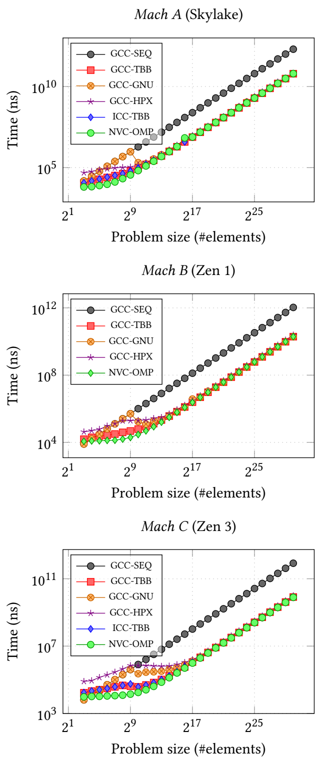
Regarding the differences between the backends, the NVIDIA compiler with the OpenMP backend is the fastest in almost every scenario and machine, except for extremely small problem sizes. This difference is particularly noticeable with problem sizes up to 2^{20} or 2^{25} elements depending on the system. The higher the number of threads, the bigger the difference between NVC-OMP and the rest of the backends, as shown in Figure 3. The TBB backend sits in the middle, with a performance that is consistent regardless of the compiler used, GCC or ICC. The GNU backend (which uses OpenMP) is often slightly slower than the TBB backend. It is worth noting that the GNU backend uses a parallel execution only for problem sizes of 2^{10} elements or larger, using the sequential version for smaller inputs. Finally, the HPX backend is the slowest in almost every scenario, with a performance that is consistently worse than the rest of the backends. Additionally, as shown in Figure 3, the HPX backend has poor scalability, with a speedup that is almost constant in systems with more than 16 threads.

To assess the performance differences between the backends, we utilized Likwid to gather performance counters data for each backend. Likwid's report (Tab. 3) indicates that HPX produces more instructions than the other backends, up to 147 % more than the ICC-TBB backend, suggesting that HPX spends more time managing and scheduling the individual work chunks.

When the computational intensity is increased (using 1000 iterations per array element instead of 1 iteration), all compilers and backends are much closer in performance. Figure 2 shows only

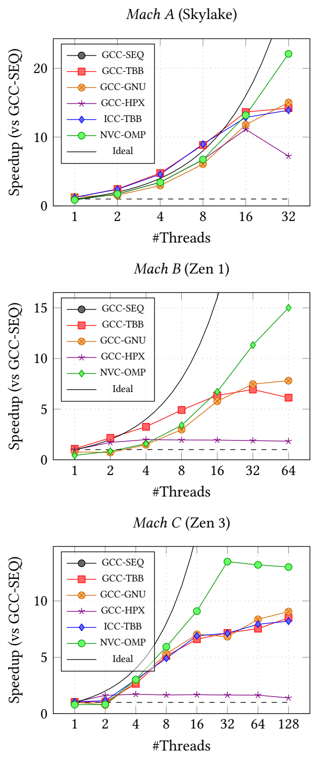


(a) $k_{\text{it}} = 1$.

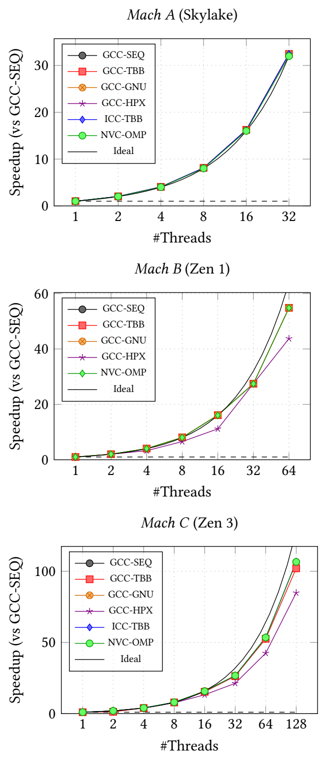


(b) $k_{\text{it}} = 1000$.

Figure 2: Results for benchmark `X::for_each`. Problem scaling using all cores except for GCC's seq. implementation. Lower is better.



(a) $k_{\text{it}} = 1$.



(b) $k_{\text{it}} = 1000$.

Figure 3: Results for benchmark `X::for_each`. Strong scaling with 2^{30} elements. Higher is better.

Table 3: Executed instructions in 100 calls to the function `X::for_each` ($k_{\text{it}} = 1$) on *Mach A* (Skylake).

Metric	GCC TBB	GCC GNU	GCC HPX	ICC TBB	NVC OMP
Instructions	1.72T	2.41T	3.83T	1.55T	2.24T
FP scalar	107G	107G	107G	107G	107G
FP 128-bit packed	0	0	0	0	0
FP 256-bit packed	0	0	0	0	0
GFLOP/s	5.41	6.51	4.06	5.02	7.26
Mem. bandwidth (GiB/s)	107.6	116.6	75.6	104.5	119.1
Mem. data volume (GiB)	2128	1925	1850	2151	1762

significant differences for small problem sizes, i.e., instances with up to 2^{10} elements on *Mach A* (Skylake) and *Mach B* (Zen 1), and up to 2^{16} elements on *Mach C* (Zen 3).

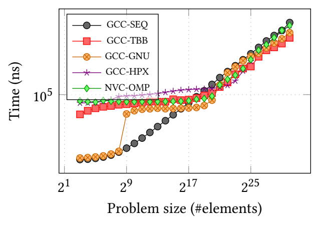
As shown in Figure 3, the speedup is close to the ideal speedup for all the backends and compilers, except for the HPX backend, which has a slightly worse performance than the rest of the backends. For example, on *Mach C* (Zen 3), HPX achieves a maximum speedup of 84.8 with 128 cores, while the rest of the backends achieve a speedup between 102.0 and 106.7. These values correspond to a parallel efficiency of 66 % for HPX, while for the rest of the backends, the parallel efficiency is between 79 % and 83 %.

5.3 `X::find`

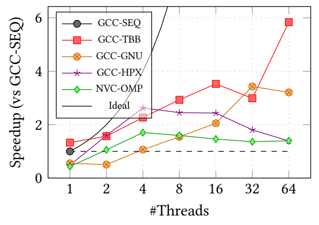
`X::find` performs a linear search on an input array that involves synchronization points between threads.

We conducted experiments on all three machines, but only show results for *Mach B* (Zen 1) in Figure 4 due to space constraints.

Figure 4a highlights the results for scaling problem sizes while consistently using 64 threads on the 64-core machine. For small problem sizes, the sequential implementation significantly outperforms the parallel version in execution times, often by orders of magnitude, which highlights the overhead required for managing parallel threads. This difference consistently decreases until problem sizes reach approximately 2^{16}, at which point the performance benefits of using the parallel version begin to compensate for the overhead. Similarly to the results shown for `X::for_each`, the GNU parallel implementation utilizes the sequential version until the problem size reaches 2^{9}, when it switches to a parallel implementation. This strategy is very effective. Since these switching points are tunable in the backend, this threshold should be adjusted for production runs on a specific target architecture.

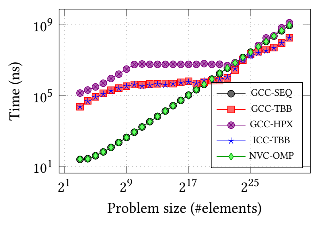


(a) Problem scaling with 64 threads. Lower is better.

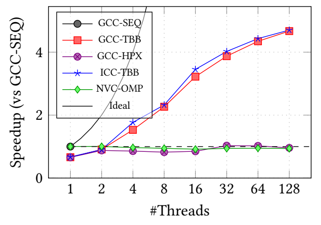


(b) Strong scaling with 2^{30} elements. Higher is better.

Figure 4: Results for `X::find` in *Mach B* (Zen 1).



(a) Problem scaling with 128 threads. Lower is better.



(b) Strong scaling with 2^{30} elements. Higher is better.

Figure 5: Results for `X::inclusive_scan` in *Mach C* (Zen 3).

For problems with more than 2^{18} elements, the parallel implementations significantly outperform the sequential version. This observation was consistent across the three machines.

The speedup analysis, shown in Figure 4b, reveals that the resulting parallel efficiency is relatively low. The maximum speedup achieved by any backend was about 6 with GCC-TBB and 64 threads on *Mach B* (Zen 1). This observation is not surprising, as `X::find` is strongly memory-bound. A previous bandwidth analysis with the STREAM benchmark [17] revealed that a speedup of approximately 7 can be expected when using all cores instead of just one (cf. Table 2).

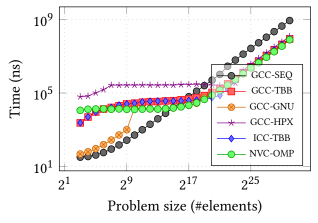
5.4 `X::inclusive_scan`

We have also evaluated the performance of the various backends for the `inclusive_scan` function, which computes an inclusive prefix sum for a specific binary operation. We used the `std::plus<>()` operation, which is the default. We present the results for the problem and strong scaling experiments on *Mach C* (Zen 3) in Figure 5.

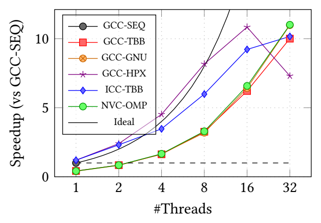
We omit the results of GCC-GNU since the GNU's collection of parallel algorithms does not implement a parallel `inclusive_scan` function. Additionally, the NVIDIA compiler with the OpenMP backend currently does not support this operation, falling back to the sequential implementation.

The performance results of the problem scaling experiments with `X::inclusive_scan` are summarized in Figure 5a. We can observe that the sequential implementations, including NVC-OMP, outperform the parallel implementations for problem sizes up to 2^{22} double elements, which is the L2 cache size of the *Mach C* (Zen 3). When the problem size exceeds the last level cache capacity (2^{26} double elements), the parallel backends are significantly faster than the sequential implementations, with GCC-HPX being an exception.

Similar results can also be seen in Figure 5b, where the number of threads is increased until 128. While NVC-OMP and GCC-HPX backends do not provide any scaling on this machine, the TBB-based implementations with GCC or ICC reduce the running time monotonically. Nonetheless, the parallel efficiency is relatively low, as the TBB-based implementations only achieve a speedup of about 5 with 128 threads. This result is also expected due to the memory-bound nature of `X::inclusive_scan`. Consequently, the performance is limited by the number of memory controllers accessible to the threads.



(a) Problem scaling with 32 threads. Lower is better.



(b) Strong scaling with 2^{30} elements. Higher is better.

Figure 6: Results for `X::reduce` in *Mach A* (Skylake).

5.5 `X::reduce`

The `X::reduce` benchmark computes the sum of all elements in an array. This is a fundamental operation in parallel computing and is often used as a building block for more complex algorithms.

Note that the GNU's collection of algorithms does not include a `reduction` function so `accumulate` has been used instead.

Regarding the problem scaling (Figure 6a), the same pattern as in the previous benchmarks is observed. The sequential implementation is faster for small problem sizes, up to 2^{15} elements, and then the parallel implementations start to compensate for the overhead.

The speedup, shown in Figure 6b, divides the backends into two groups. The NVC-OMP, GCC-TBB, and GCC-GNU backends provide very similar results. Paying attention to the sequential performance, the produced code is not as efficient as the purely sequential implementation of GCC. This hinders the speedup, which is significantly lower than the ideal speedup. The ICC-TBB and HPX backends are in the second group, with good scalability up to 16 threads. Further than that, the performance is not as good, since two NUMA nodes are used, and the backends are not able to manage the data traffic efficiently. The drop in performance is more noticeable in the HPX backend.

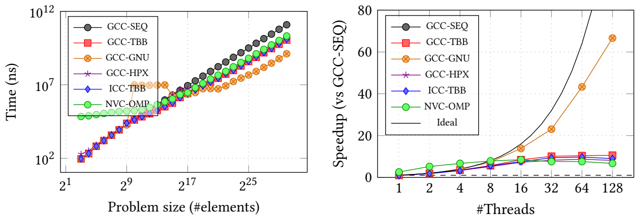
HPX produces the largest number of instructions, up to six times more than other backends, as shown in Table 4. Nevertheless, HPX also makes use of 128-bit and 256-bit vector instructions, similar to ICC. The rest of the backends do not use vector instructions, which is reflected in the number of floating-point operations per second.

5.6 `X::sort`

`X::sort` is a fundamental operation in computer science, challenging to parallelize efficiently due to the synchronization and

Table 4: Executed instructions in 100 calls to the function `X::reduce` on *Mach A* (Skylake).

Metric	GCC TBB	GCC GNU	GCC HPX	ICC TBB	NVC OMP
Instructions (any)	188G	227G	1.74T	107G	295G
FP scalar	107G	107G	472K	1.33M	107G
FP 128-bit packed	0	0	12.8K	0.56M	0
FP 256-bit packed	0	0	26G	26G	0
GFLOP/s	6.88	7.25	6.88	10.3	7.01
Mem. bandwidth (GiB/s)	75.1	58.2	65.1	97.5	56.6
Mem. data volume (GiB)	1.17	0.86	0.90	1.12	0.87



(a) **Problem scaling with 32 threads. Lower is better.**

(b) **Strong scaling with 2^{30} elements. Higher is better.**

Figure 7: Results for `X::sort` in *Mach C* (Zen 3).

communication required between threads. This is reflected in the results, as shown in Figure 7.

We monitored the CPU usage and found that, similarly to the GNU backend, the TBB implementation falls back to a sequential execution for small problem sizes, up to 2^9 elements. Likewise, HPX delegates the work to a single thread with input sizes of 2^{15} or smaller, making use of parallelism for larger inputs.

The speedup analysis reveals two interesting behaviors. First, the NVC-OMP backend is the fastest for a small number of threads due to a more efficient use of the L2 cache. Second, the GNU backend is the most efficient for a larger number of threads, thanks to a higher bandwidth, suggesting a better placement of threads and data. The other backends exhibit poor scalability, achieving a speedup that is far from ideal.

5.7 Summary and Additional Remarks

Table 5 summarises the results obtained on the three parallel machines, showing the speedup obtainable with all available cores compared to a fixed baseline performance. We selected the sequential implementation of GCC as the baseline, which may result in speedups exceeding the total core count.

Generally, the parallel implementations are faster than the sequential, however, the speedup is usually far from ideal when using the maximum number of threads that the machine provides. Most of the algorithms present a speedup close to 10, being this value consistent across the different machines. Even in the `X::for_each` algorithm, being an embarrassingly parallel problem, the speedup is highly dependent on the computational load. For high computational intensity, the speedup is close to the ideal because the computational load is the leading factor. For low computational intensity, the overhead of the parallelization is more significant, resulting in a poor speedup in most cases.

With the poor scalability of the parallel implementations, efficiency must be considered. Table 6 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. The results indicate that backends typically fail to handle more than 16 threads efficiently. It should be noted that this number matches the cores per NUMA node in *Mach A* (Skylake) and *Mach C* (Zen 3). This suggests that the backends are not able to manage the data/thread placement efficiently when dealing with more than one NUMA node.

The sizes of the binary files generated by the different compilers and backends are shown in Table 7. The internal complexity of the backends is reflected in the binary sizes, with the HPX backend producing the largest binaries, up to 62 MiB. Next, the TBB backend produces binaries of 16.64 MiB and 17.21 MiB for ICC and GCC, respectively. The GNU backend produces binaries of 5.31 MiB, double the size of sequential binaries of GCC, 2.52 MiB. The NVIDIA compiler produces remarkably small binaries, 1.81 MiB and 7.80 MiB for the OpenMP and CUDA versions, respectively.

5.8 Performance on GPUs

The last set of experiments is focused on the performance of the parallel algorithms on GPUs, using the NVIDIA HPC SDK with the CUDA compiler. Since GPUs are traditionally optimized for `float` (32-bit) operations, we carried out additional experiments using this data type. Also, the most interesting algorithms for the GPUs are the `X::for_each` and the `X::reduce` since, given their structure, the GPUs can exploit their architectural characteristics.

It should be mentioned that the keyword `volatile` (see Listing 1) does not produce any error when compiling the code with the NVIDIA compiler targeting the GPU, but it seems to be ignored. Thus, when the number of iterations is known at compile time, the loop is optimized away. Interestingly, this optimization is always performed for `int`, but only for `double` when the number of iterations is less than 65 001 (suggesting the presence of a magic number in the compiler). For 32-bit floats, the loop is never optimized.

An important aspect to consider when using GPUs is the communication between the host and the device. Since the GPUs have their own memory, data must be transferred between the host and the device. The CUDA backend uses *Unified Memory* to automatically manage the data transfers by copying memory pages on demand. This approach simplifies the code but might lead to a significant overhead when the data is frequently accessed by both the host and the device. Two factors might alleviate this overhead: first, the kernel's computational intensity should be high enough, so this is the leading factor in the execution time; second, consecutive calls to the GPU with the same data will reduce the overhead of the memory transfers, as the data will be already in the GPU's memory.

We explore the first factor by analyzing the performance of the `X::for_each` algorithm with different computational intensities, as shown in Figure 8. In this case, we enforced the data transfer back

Table 5: Speedup against GCC's sequential implementation for machines *Mach A* (Skylake), *Mach B* (Zen 1), and *Mach C* (Zen 3), with 32, 64, and 128 cores, respectively. Notation is *Mach A* |*Mach B* |*Mach C*. Problem size is 2^{30}. Higher is better.

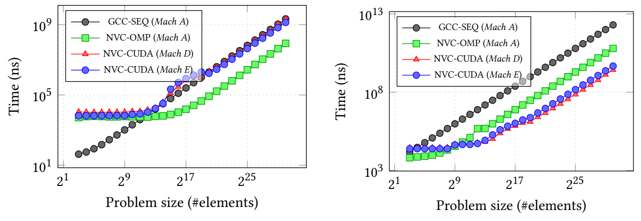
	`X::find`	`X::for_each` $k_{\text{it}} = 1$	`X::for_each` $k_{\text{it}} = 1000$	`X::inclusive_scan`	`X::reduce`	`X::sort`
GCC-TBB	8.9 | 5.8 | 4.7	14.2 | 6.1 | 8.5	32.5 | 54.9 | 102.0	4.5 | 3.1 | 4.7	10.0 | 5.1 | 6.9	9.7 | 9.4 | 10.6
GCC-GNU	8.0 | 3.2 | 2.2	15.0 | 7.8 | 9.1	32.5 | 54.9 | 106.5	N/A | N/A | N/A	11.0 | 4.7 | 6.0	25.4 | 26.9 | 66.6
GCC-HPX	6.4 | 1.4 | 1.1	7.2 | 1.8 | 1.4	32.4 | 43.7 | 84.8	3.0 | 0.9 | 1.0	7.3 | 0.9 | 1.2	10.1 | 8.0 | 8.1
ICC-TBB	9.0 | N/A | 4.8	13.9 | N/A | 8.2	32.5 | N/A | 106.7	4.5 | N/A | 4.7	10.2 | N/A | 6.8	10.1 | N/A | 9.0
NVC-OMP	6.1 | 1.4 | 1.2	22.1 | 15.0 | 13.0	32.0 | 54.8 | 106.5	0.9 | 0.8 | 0.9	11.0 | 4.8 | 11.9	7.1 | 6.3 | 6.7

Table 6: Maximum number of threads such that efficiency is above 70 % (compared to the seq. execution) for machines *Mach A* (Skylake), *Mach B* (Zen 1), and *Mach C* (Zen 3). Notation is *Mach A* |*Mach B* |*Mach C*. Problem size is 2^{30}. Higher is better.

	`X::find`	`X::for_each` $k_{\text{it}} = 1$	`X::for_each` $k_{\text{it}} = 1000$	`X::inclusive_scan`	`X::reduce`	`X::sort`
GCC-TBB	2 | 1 | 2	8 | 4 | 1	32 | 64 | 128	4 | 1 | 1	32 | 8 | 16	8 | 8 | 8
GCC-GNU	8 | 1 | 1	1 | 1 | 1	32 | 64 | 128	N/A | N/A | N/A	32 | 8 | 16	32 | 16 | 32
GCC-HPX	16 | 4 | 1	16 | 2 | 2	32 | 32 | 16	4 | 2 | 1	8 | 2 | 4	4 | 2 | 4
ICC-TBB	2 | N/A | 1	8 | N/A | 4	32 | N/A | 128	4 | N/A | 1	4 | N/A | 1	8 | N/A | 8
NVC-OMP	16 | 4 | 4	32 | 32 | 16	32 | 64 | 128	1 | 1 | 1	32 | 16 | 32	2 | 2 | 2

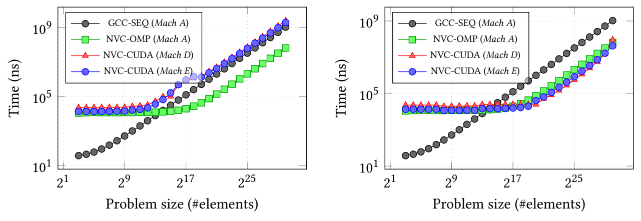
Table 7: Binary sizes for the different compilers and backends used in the experiments in *Mach A* (Skylake) and *Mach D* (Tesla). Lower is better.

Target machine			*Mach A*				*Mach D*
Compiler Backend	GCC SEQ	GCC TBB	GCC GNU	GCC HPX	ICC TBB	NVC OMP	NVC CUDA
Bin. size (MiB)	2.52	17.21	5.31	61.98	16.64	1.81	7.80



(a) $k_{\text{it}} = 1$ (b) $k_{\text{it}} = 1000$

Figure 8: Results for `X::for_each`. Problem scaling using all cores except for GCC's seq. implementation. Data type: `float`. Lower is better.



(a) With GPU-to-Host transfer **(b) Without GPU-to-Host transfer**

Figure 9: Results for `X::reduce`. Problem scaling using all cores except for GCC's seq. implementation. Data type: `float`. Lower is better.

to the host between each call to the GPU to evaluate the overhead of the memory transfers and how it can be mitigated by increasing the workload. When the intensity is low the performance is limited by the overhead of the memory transfers. The cost of launching a kernel and the communication is so high that the GPU is slower than the CPUs in parallel, and even in some cases, slower than the sequential implementation. However, when the workload is increased, the massively parallel architecture of the GPUs allows them to outperform the parallel CPU implementation by a factor 23.5× on *Mach D* (Tesla) and 13.3× on *Mach E* (Ampere).

The effect of chaining multiple calls to the GPU is shown in Figure 9. On one hand, when data is constantly transferred back from the GPU to the host, the execution time is communication-limited, up to a point where the GPUs are slower than the CPU with sequential implementation. On the other hand, when data is already in the device's memory, performance improves enough to outperform the CPUs.

As a final remark, it is important to note that the input size is a critical factor in any scenario because launching a kernel is a costly operation that is not amortized for small problem sizes.

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 benchmark suite, we aim to enable users to compare compilers and backends, helping them decide which best suits the specific characteristics of their system.

We demonstrate the usefulness of the benchmarks by conducting a comprehensive performance analysis on current multi-core architectures, benchmarking some of the most commonly used algorithms in the STL. This analysis took into account not only the compiler and backend being used but also the input size, the number of threads used, and the memory allocation strategy.

First, we showed the potential performance improvements when using a custom parallel allocator on NUMA systems. Generally, execution times are improved, up to 63 % in the best case, but some losses can be observed in certain scenarios.

Second, by testing various input sizes, we compared the overhead of launching parallel sections and the differences between backends. This analysis highlighted the overhead of the HPX backend and revealed that GNU's implementation defaults to sequential execution for smaller inputs. For large input sizes, the parallel implementations typically outperformed their sequential counterparts. However, the parallel efficiency is far from ideal in most scenarios. Thus, computational load has to be the main concern when selecting the number of threads, as the performance of the algorithms is highly dependent on it.

Finally, we evaluated the performance of the CUDA backend for the NVIDIA compiler with two different GPUs. Not surprisingly, we found that data transfers were the main bottleneck. Therefore, users should aim to chain as many operations as possible on the GPU to minimize the number of data transfers required between the host and the device. Alternatively, they need to ensure that the computational load is sufficiently high to compensate for the overhead of the communication between host and device.

Regarding future work, we would like to expand our benchmark suite, to support more compilers and backends. Similarly, an extended analysis could include other architectures, such as ARM processors or other accelerators like FPGAs.

REFERENCES

[1] Victor Artigues, Katharina Kormann, Markus Rampp, and Klaus Reuter. 2020. Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. *Concurr. Comput. Pract. Exp.* 32, 11 (2020). https://doi.org/10.1002/CPE.5640

[2] David Beckingsale, Thomas R. W. Scogland, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, and Brian S. Ryujin. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC@SC)*. IEEE, 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

[3] Nathan Bell and Jared Hoberock. 2012. Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 359–371. https://doi.org/10.1016/B978-0-12-385963-1.00026-5

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.* 14, 3 (aug 2000), 189–204. https://doi.org/10.1177/109434200001400303

[5] J. Mark Bull, Fiona Reid, and Nicola McDonnell. 2012. A Microbenchmark Suite for OpenMP Tasks. In *Proceedings of the 8th International Workshop on OpenMP (IWOMP) (Lecture Notes in Computer Science, Vol. 7312)*, Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro (Eds.). Springer, 271–274. https://doi.org/10.1007/978-3-642-30961-8_{2}{4}

[6] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. 1994. The MPI Message Passing Interface Standard. In *Programming Environments for Massively Parallel Distributed Systems*, Karsten M. Decker and René M. Rehmann (Eds.). Birkhäuser Basel, Basel, 213–218.

[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[8] Gregor Daiß, Patrick Diehl, Hartmut Kaiser, and Dirk Pflüger. 2023. Stellar Mergers with HPX-Kokkos and SYCL: Methods of using an Asynchronous Many-Task Runtime System with SYCL. In *Proceedings of the 2023 International Workshop on OpenCL (IWOCL)*. ACM, 8:1–8:12. https://doi.org/10.1145/3585341.3585354

[9] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *Int. J. Comput. Sci. Eng.* 17, 3 (2018), 247–262. https://doi.org/10.1504/IJCSE.2018.095847

[10] Google. 2016. *Google Benchmark*. https://github.com/google/benchmark

[11] Jeff R. Hammond, Michael Kinsner, and James C. Brodman. 2019. A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications. In *Proceedings of the International Workshop on OpenCL (IWOCL)*. ACM, 15:1–15:2. https://doi.org/10.1145/3318170.3318193

[12] Leah Hoffmann. 2022. Learning new things and avoiding obstacles. *Commun. ACM* 65, 6 (2022). https://doi.org/10.1145/3530690

[13] Sascha Hunold and Klaus Kraßnitzer. 2022. A Quantitative Analysis of OpenMP Task Runtime Systems. In *Proceedings of the 14th BenchCouncil International Symposium on Benchmarking, Measuring, and Optimization (Bench 2022) (Lecture Notes in Computer Science, Vol. 13852)*, Ana Gainaru, Ce Zhang, and Chunjie Luo (Eds.). Springer, 3–18. https://doi.org/10.1007/978-3-031-31180-2_1

[14] Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin A. Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad, Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. 2020. HPX - The C++ Standard Library for Parallelism and Concurrency. *J. Open Source Softw.* 5, 53 (2020), 2352. https://doi.org/10.21105/joss.02352

[15] Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Alexander Hirsch, Nicolai Stawinoga, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. 2020. SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 26th Euro-Par 2020 (Lecture Notes in Computer Science, Vol. 12247)*, Maciej Malawski and Krzysztof Rzadca (Eds.). Springer, 629–644. https://doi.org/10.1007/978-3-030-57675-2_{3}{9}

[16] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. 2022. Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems. In *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS@SC)*. IEEE, 36–47. https://doi.org/10.1109/PMBS56514.2022.00009

[17] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[18] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[19] Simon John Pennycook, Ben Ashbaugh, James C. Brodman, Michael Kinsner, Steffen Larsen, Gregory Lueck, Roland Schulz, and Michael Voss. 2023. Towards Alignment of Parallelism in SYCL and ISO C++. In *Proceedings of the 2023 International Workshop on OpenCL (IWOCL)*. ACM, 14:1–14:9. https://doi.org/10.1145/3585341.3585371

[20] S. John Pennycook and Jason D. Sewall. 2021. Revisiting a Metric for Performance Portability. In *International Workshop on Performance, Portability and Productivity in HPC (P3HPC@SC)*. IEEE, 1–9. https://doi.org/10.1109/P3HPC54578.2021.00004

[21] Simon J. Pennycook, Jason D. Sewall, and Victor W. Lee. 2019. Implications of a metric for performance portability. *Future Gener. Comput. Syst.* 92 (2019), 947–958. https://doi.org/10.1016/j.future.2017.08.007

[22] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly. http://www.oreilly.com/catalog/9780596514808/index.html

[23] Ruymán Reyes, Gordon Brown, Rod Burns, and Michael Wong. 2020. SYCL 2020: More than meets the eye. In *Proceedings of the International Workshop on OpenCL (IWOCL)*, Simon McIntosh-Smith (Ed.). ACM, 4:1. https://doi.org/10.1145/3388333.3388649

[24] Johannes Singler, Peter Sanders, and Felix Putze. 2007. MCSTL: The Multi-core Standard Template Library. In *Proceedungs of the 13th Euro-Par (Lecture Notes in Computer Science, Vol. 4641)*, Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol (Eds.). Springer, 682–694. https://doi.org/10.1007/978-3-540-74466-5_{7}{2}

[25] J. Treibig, G. Hager, and G. Wellein. 2010. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. San Diego CA.

[26] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Q. Dang, Nathan D. Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan R. Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah J. Wilke. 2022. Kokkos 3: Programming Model

Extensions for the Exascale Era. *IEEE Trans. Parallel Distributed Syst.* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[27] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.