What You Always Wanted To Know About C++ Performance Portability (But Were Afraid to Do)

Austrian-Slovenian HPC Meeting 2024 — ASHPC24

Ruben Laso, Diego Krupitza, and Sascha Hunold
{laso, krupitza, hunold}@par.tuwien.ac.at

June 12, 2024

Research Group for Parallel Computing, TU Wien

C++ in High-Performance Computing

- Languages like C, C++ and Fortran are the most used in HPC
  - Existing code base
  - Performance
  - Compatibility with new hardware
- C++ can be used in several types of architectures
  - Multi-core and many-core CPUs $\rightarrow$ OpenMP, TBB
  - GPUs $\rightarrow$ CUDA, OpenCL
  - FPGAs $\rightarrow$ SYCL
- Should we use a different code for each system?

C++ STL in HPC

Performance Portability

Same code performs “well” on different architectures

Performance Portability in C++

- Several libraries: Kokkos, RAJA, HPX, ...
- **C++17** with **execution policies**
  - Parallel execution of the algorithms in STL (standard library)
  - Different compilers/backends

C++ STL in HPC

Questions

- **Speedup and efficiency** of parallel STL algorithms?
- Which is the **best compiler/backend**? GCC vs ICC, TBB vs OpenMP, ...
- **GPUs** performance?

Contributions

- *pSTL-Bench*: micro-benchmark suite
- Evaluation of the performance for
  - Different compilers: GCC, ICC, NVIDIA HPC SDK
  - Different backends: OpenMP, TBB, HPX, CUDA
  - Different systems: Intel and AMD CPUs, NVIDIA GPUs

pSTL-Bench

- Code available on github.com/parlab-tuwien/pSTL-Bench
- Suite of **micro-benchmarks** to test performance portability in C++
  - STL algorithms: `std::for_each`, `std::reduce`, `std::sort`, ...
- Features:
  - Number of threads: with `OMP_NUM_THREADS` or `--hpx::threads=N`
  - HW Perf. Counters: PAPI's HL or Likwid's Marker APIs
  - Different (customizable) input sizes and data types
  - Custom NUMA allocator

Experiments

Variables

- Input sizes: 2^3 to 2^{30} elements $\rightarrow$ 64B to 8GB
- Data type: `double` and `float`
- *pSTL-Bench*'s NUMA allocator
- No control of thread and memory placement

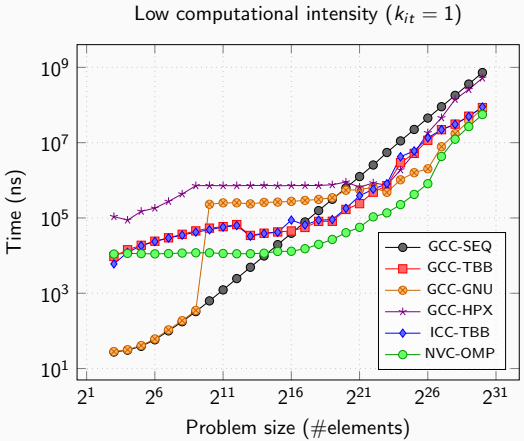
Contenders

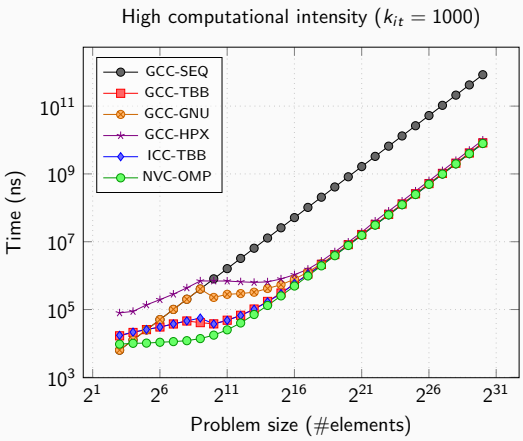
- Compilers: GCC, ICC, NVIDIA HPC SDK
- Backends: GNU (OpenMP), TBB, HPX, Thrust (OMP), CUDA

Experiments

Machine	**VSC-5 (Zen 3)**	**Hydra (Skylake)**	**Tesla**	**Ampere**
CPU/GPU	**AMD EPYC 7713**	**Intel Xeon 6130F**	**NVIDIA Tesla T4**	**NVIDIA Ampere A2**
Architecture	Zen 3	Skylake	Turing	Ampere
Sockets | NUMA nodes	2 | 8	2 | 2	1 | 1	1 | 1
Total #cores | threads	128 | 256	32 | 32	2560 | 2560	1280 | 1280
Max. #threads used	**128**	**32**	**2560**	**1280**
Memory (node / GPU)	512 GiB	48 GiB	16 GiB	8 GiB
Memory (per core)	4 GiB	1.5 GiB	—	—
STREAM BW 1 | all core(s) (GB/s)	42.6 | 249	11.7 | 135	N/A | 264	N/A | 172

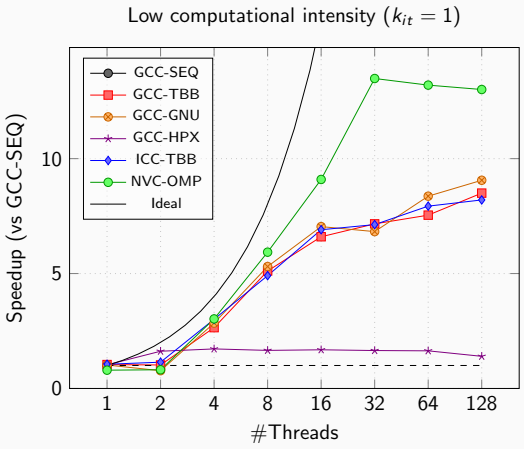
Results - Execution time scaling

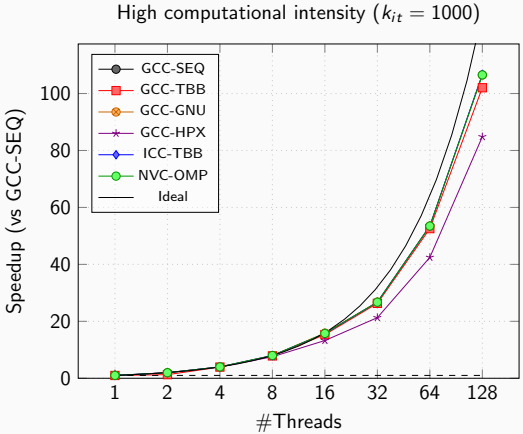




Execution time scaling of `for_each` in **VSC-5 (Zen 3)**. Data type: `double`. **All cores** are used except for GCC-SEQ. Lower is better.

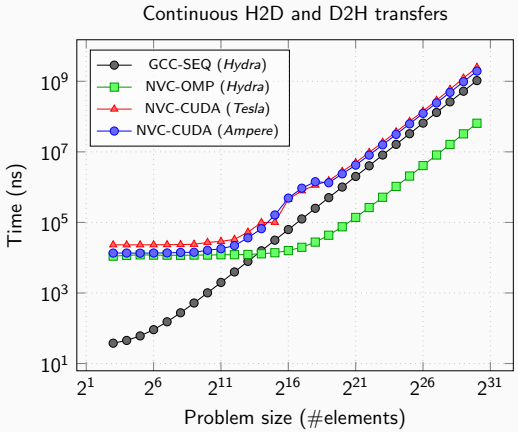
Results - Speedup

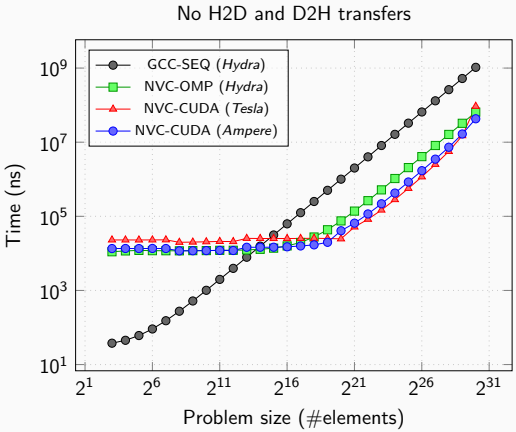




Strong scaling of `for_each` with 2^{30} `doubles` in **VSC-5 (Zen 3)**. Higher is better.

Results - GPUs





Execution time scaling of `reduce`. Data type: `float`. **All cores** are used except for GCC-SEQ. Lower is better.

Get your hands on it

Code and papers

- *pSTL-Bench*: github.com/parlab-tuwien/pSTL-Bench
- arXiv preprint: https://arxiv.org/abs/2402.06384
- ICPP 2024 paper: incoming

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

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

What You Always Wanted To Know About C++ Performance Portability (But Were Afraid to Do)

Austrian-Slovenian HPC Meeting 2024 — ASHPC24

Ruben Laso, Diego Krupitza, and Sascha Hunold
{laso, krupitza, hunold}@par.tuwien.ac.at

June 12, 2024

Research Group for Parallel Computing, TU Wien

Additional content

Maximum number of threads such that **efficiency is above** 70 % (compared to the seq. execution) for **VSC-5 (Zen 3)**. Problem size is 2^{30}. Higher is better.

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

Additional content

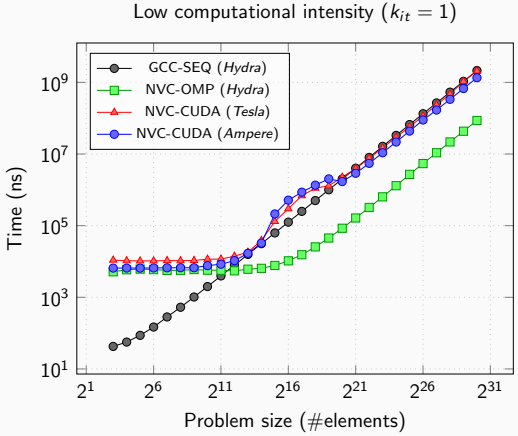
Executed instructions in 100 calls to `std::for_each` ($k_{\text{it}} = 1$) on **Hydra (Skylake)**.

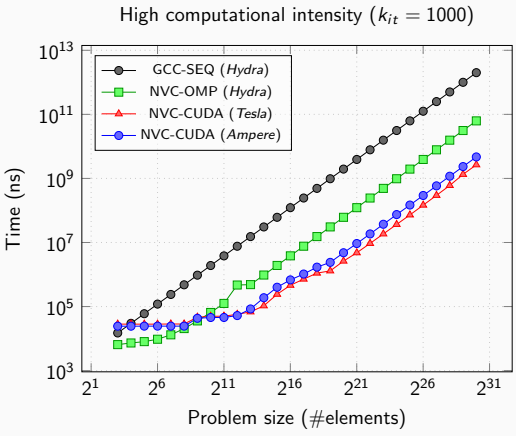
Metric	GCC TBB	GCC GNU	GCC HPX	ICC TBB	NVC OMP
Instructions	1.72T	2.41T	3.83T	1.55T	2.24T
FP scalar	107G	107G	107G	107G	107G
FP 128-bit packed	0	0	0	0	0
FP 256-bit packed	0	0	0	0	0
GFLOP/s	5.41	6.51	4.06	5.02	7.26
Mem. bandwidth (GiB/s)	107.6	116.6	75.6	104.5	119.1
Mem. data volume (GiB)	2128	1925	1850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

Binary sizes in **Hydra (Skylake)** and **Tesla**. Lower is better.

Compiler Backend	Binary size (MiB)
GCC-SEQ	2.5
GCC-TBB	17.2
GCC-GNU	5.3
GCC-HPX	62.0
ICC-TBB	16.6
NVC-OMP	1.8
NVC-CUDA	7.8

Additional content





Execution time scaling of `for_each`. Data type: `float`. All cores are used except for GCC-SEQ. Lower is better.