2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2) | 978-1-6654-1140-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ESPM254806.2021.00008

Parallel SIMD - A Policy Based Solution for Free Speed-Up using C++ Data-Parallel Types

Srinivas Yadav
Keshav Memorial Institute of Technology,
Hyderabad, India
vasu.srinivasvasu.14@gmail.com

Nikunj Gupta
University of Illinois at Urbana-Champaign
Illinois, U.S.A.
nikunj@illinois.edu

Auriane Reverdell
Swiss National Supercomputing Centre
Zurich, Switzerland
aurianer@cscs.ch

Hartmut Kaiser
Center for Computation Technology
Louisiana State University
Baton Rouge, U.S.A.
hkaiser@cct.lsu.edu

***Abstract*—Recent additions to the C++ standard and ongoing standardization efforts aim to add data-parallel types to the C++ standard library. This enables the use of vectorization techniques in existing C++ codes without having to rely on the C++ compiler's abilities to auto-vectorize the code's execution. The integration of the existing parallel algorithms with these new data-parallel types opens up a new way of speeding up existing codes with minimal effort. Today, only very little implementation experience exists for potential data-parallel execution of the standard parallel algorithms. In this paper, we report on experiences and performance analysis results for our implementation of two new data-parallel execution policies usable with HPX's parallel algorithms module: `simd` and `par_simd`. We utilize the new experimental implementation of data-parallel types provided by recent versions of the GCC and Clang C++ standard libraries. The benchmark results collected from artificial tests and real-world codes presented in this paper are very promising. Compared to sequenced execution, we report on speed-ups of more than three orders of magnitude when executed using the newly implemented data-parallel execution policy `par_simd` with HPX's parallel algorithms. We also report that our implementation is performance portable across different compute architectures (x64 – Intel and AMD, and Arm), using different vectorization extensions (AVX2, AVX512, and NEON128).**

***Index Terms*—Vectorization, SIMD, Asynchronous Many-Task Systems, HPX**

I. Introduction

The C++ parallel algorithms that were added to the latest C++17 and C++20 standards provide a convenient and well integrated API for integrating parallel operations with existing C++ codes [1], [2]. The new parallel algorithms are specified to expect to be invoked using an *execution policy* as their first argument. The C++ standard specifies several execution policies (sequenced, parallel, and unsequenced execution), but allows for implementation of more policies in the future. All existing mainstream C++ compilers and libraries have started to implement these algorithms and the related execution policies with their recent releases. That has significantly raised the awareness of the community, and we see an uptake in the use of the new parallel algorithms in real world codes.

Additional recent standardization efforts have proposed specifying C++ data types supporting data parallelism, i.e., *vectorization* (see Parallelism TS V2, N4755 [3]). These data types directly represent the architecture's vector registers and implement all necessary operations for those (e.g., load and store operations, arithmetic operations, special math functions, masking, etc.). This specification aims for a straightforward conversion of existing non-vectorized C++ codes to take advantage of the architecture's data-parallel operations by simply replacing the utilized data types (e.g., use `std::experimental::simd<float>` instead of a plain `float`). First implementations of the proposed data-parallel types have been experimentally added to recent versions of at least two of the C++ standard libraries: stdlibc++ (since GCC 11 [4]) and libc++ (since Clang 12 [5]).

Separate standardization proposals that are being discussed currently suggest adding a special execution policy `std::execution::simd` (see P0350 [6]). This execution policy changes the behavior of a parallel algorithm such that the iteration function is not invoked with a single element of the input sequence anymore (as is the case for the existing execution policies). It internally loads an appropriate number of adjacent elements of the input sequence into an instance of a data-parallel C++ type and invokes the iteration function using that instance. As a result, the iteration function itself can be fully vectorized without having to rely on the C++ compiler's ability to apply auto-vectorization.

In this paper, we rely on HPX (see Section III-A) – the standards C++ library for parallelism and concurrency [7], [8]. HPX is an open source library published under the very liberal Boost Software License [9] enabling its use in any context. HPX provides a full set of the standard parallel algorithms, which makes it a perfect target for experimental implemen-

978-1-6654-1140-0/21/$31.00 ©2021 IEEE
DOI 10.1109/ESPM254806.2021.00008

tations of new standardization proposals. We implemented two new execution policies, the **simd** policy as proposed by P0350 [6] (sequenced, vectorized execution) and a related parallel version **par_simd** that comprises an HPX specific extension additionally utilizing the available parallelization infrastructure (i.e., vectorization *and* parallel execution).

We make the following key contributions through our work:

1) The extension of the proposed work in HPX beyond N4755 [3], i.e., the **par_simd** execution policy, and our implementation experiences of the policies proposed in the paper.
2) Implementation of policies that significantly lower the work-effort of adding explicit-vectorization while providing free speed-ups.
3) The collected performance measurement results obtained from applying different execution policies to a set of artificial benchmarks and real-world applications.
4) An in-depth comparison of the collected performance results on different compute architectures, namely x64 (Intel and AMD) and ARMv8/ARMv8.2 (Arm).

The paper is structured as follows. Section II revisits auto-vectorization through **#pragma** directives and how our work differs from prior works. Section III discusses various concepts that are utilized in the later sections. Section IV elaborates on the benchmarks with code snippets to better explain the decisions made while selecting the necessary functions. Section V briefly summarizes the systems on which the benchmarks were run, the testing strategy, and the dependencies on which the test code relies on. Finally, Section VI explains the results obtained on different architectures and Section VII draws conclusion on these results.

II. Related Work

Compiler based auto-vectorization [10]–[13] techniques, including but not limited to **#pragma** directives such as OpenMP **#pragma** [14], [15] are widely used for loop-vectorization. ISO C++17 and C++20 introduces two new execution policies related to vectorization, namely **unseq** and **par_unseq** [1], [2] that abstracts these techniques to provide auto loop-vectorization. However, these techniques only provide compilers a hint, i.e., the compiler makes the final decision on vectorization.

Our work differs from the said description by introducing support for using explicit data-parallel types (the Parallelism TS V2 [3]) with the standard parallel algorithms, allowing the user to take advantage of data-locality based algorithms while ensuring an explicitly vectorized code that guarantees speed-ups. To achieve the same, we make use of the data-parallel types to create two new execution policies, namely **simd** and **par_simd**, that ensures explicit vectorization with both sequential and parallel codes.

III. Background

A. HPX

HPX [7], [8] is a ISO C++ standards conforming library for parallel and distributed computing built on top of an asynchronous many-task (AMT) runtime system. Exposing ISO C++ standards API enables wait-free asynchronous parallel programming through futures, channels, and other synchronization primitives in HPX. Tasking provides a means to fully exploit available parallelism (through concurrency) on complex, emerging HPC architectures.

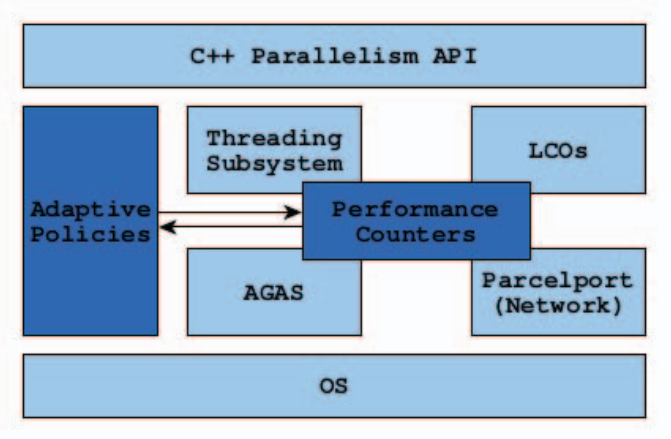

Fig. 1: Outline of HPX's architecture

HPX can be divided into four main components. The Active Global Address Space (AGAS) is used to track remote objects, which is achieved through a unique Global Identifier (GID) assigned to each HPX object. Furthermore, AGAS enables HPX to load balance through object migration. The threading sub-system comprises lightweight threads and tasking APIs on top of it. These threads are scheduled on top of OS threads, reducing overheads relating to creation, synchronizations and context switching. Local Control Objects (LCOs) is a class of synchronization functions that facilitates the user to synchronize tasks within the application. Finally, the Parcelport subsystem enables migration of objects from one locality to the other through networking. This paper focuses on HPX's single node capabilities and in particular on its parallelization and vectorization techniques.

B. Vectorization

Vectorization [16], [17] uses special vector registers to allow in-core parallelism for operations on array-like types of data to speed up execution times. The size of those registers is expected to grow in the future. Vectorization leads to significant speed-ups for most compute-bound problems and slightly enhances performance on memory-bound problems. AMT systems on the other hand work across core parallelism, making vectorization based parallelism orthogonal and complementary to AMT programming. This ensures an enhanced performance as long as the I/O bandwidth allows for the additional memory loads.

Compilers have come a long way to achieve auto-vectorization using **#pragma** directives [10]–[13], but they still lack the flexibility and performance made available through explicit vectorization libraries [3], [18]–[21]. Furthermore, the compiler may not auto-vectorize loops with complicated conditional paths. Our work relies on the proposed

explicit data-parallel types [3] **std::experimental::simd** that works in tandem with the novel execution policies, namely **simd** and **par_simd**.

C. Roofline Model

The Roofline Model [22]–[24] provides bounds on estimate performances using the bottlenecks and bounds of the machine and the application. The performance is expressed using two metrics, namely, peak Computational Performance (CP) of the architecture and the peak I/O Bandwidth (BW) coupled with the Arithmetic Intensity (AI) of the application. Computational Performance is the maximum floating-point operation count that the processor can achieve per second while I/O Bandwidth is the maximum amount of memory transfer achievable by DRAM in a second. AI for an application is the number of floating-point operations executed per byte accessed from the main memory in FLOPs/Bytes.

The model identifies either CP or I/O BW as limiting factors for an application performance and provides an attainable performance through the above defined parameters as:

$$Attainable\ Performance = min(CP, AI * BW) \quad (1)$$

D. Mandelbrot

The Mandelbrot set is a set of complex numbers c, which follows the recursive equation $f(z) = z^2 + c$ with z = 0 as initial condition. The computation involved typically have a high Arithmetic Intensity revealing a compute-bound kernel. Therefore, it is interesting to analyze the performance characteristics with vectorized implementation. Figure 2[1] depicts the Mandelbrot pattern generated through the computation described before.

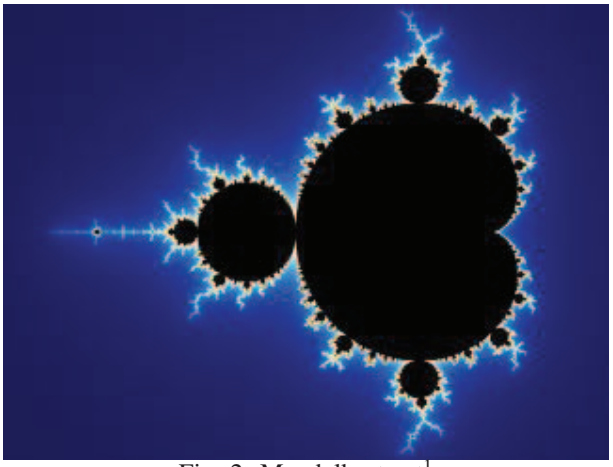

Fig. 2: Mandelbrot set[1]

E. Implementation

We implemented the **hpx::execution::simd** policy leveraging the C++ standard implementation of the data-parallel types. We specified the iteration function of the HPX's iterative algorithms for the **simd** policy to be invoked with a pack of elements (grouped in a data-parallel type) instead of a single scalar element in the case of a non-SIMD policy.

We also implemented the **hpx::execution::par_simd**. This policy is a combination of the **simd** policy and the **par** policy (the parallel policy used to dispatch to the parallel implementation of the algorithms). The newly implemented **par_simd** policy allows, in addition to the parallel policy dividing the work into chunks to be executed in parallel, the vectorization of each of those chunks. The parallel implementations being already available in HPX, we added a specialization for the **par_simd** policy to use the iteration function adapted to data-parallel types.

We finally adapted the iterative algorithms and some SIMD reduction based parallel algorithms to support the newly implemented **simd** and **par_simd** policies.

IV. Benchmarks

This section discusses the set of artificial codes that were implemented as part of the test suite to contrast between the performance and implementations of various execution policies. Later on, we use a real world example in the form of a Mandelbrot set to compare performance in a more realistic way.

A. Artificial Codes

We constructed benchmarks for two classes of algorithms, namely iterative algorithms and algorithms based on SIMD reductions.

In iterative algorithms, each element of the underlying array is mapped to a function, the result of which is stored on either the same array or a different array. For this class, we benchmark **hpx::for_each** and **hpx::transform** algorithms with both compute-bound and memory-bound kernels. The kernels were chosen based on their Arithmetic Intensity to analyze and predict possible performance improvements. The compute-bound kernel performs sine and cosine operations on the elements. The memory-bound kernel performs a saxpy operation. The sine and cosine operations were chosen based on their high Arithmetic Intensity and their efficient vectorized implementation (see Section VI) in GCC 11.1. For algorithms based on SIMD reductions, we utilize the **hpx::count** and the **hpx::find** algorithm with an emphasis on conditional statements. We run the benchmarks with an array size of 128 million elements.

```
1hpx::for_each(policy, nums.begin(), nums.end(),
2    [](auto& x)
3    {
4        for (int i = 0; i < 100; i++)
5            x = 5 * sin(x) + 6 * cos(x);
6    });
```

Listing 1: **for_each** algorithm with compute-bound kernel.

Listing 1 shows the use of the **for_each** algorithm with a compute-bound kernel that takes a generic type as its first argument. The generic type allows us to pass both data-parallel and scalar types, effectively hiding the specialization. ISO C++ math header in GCC 11.1 provides implementation for

[1]Image Credits: Wikipedia - https://en.wikipedia.org/wiki/Mandelbrot_set

both **sin** and **cos** functions for scalar and data-parallel types. Therefore, the final code snippet looks similar to a scalar implementation but supports data-parallel types as well. In the rest of the artificial codes, not displayed here, we use the block allocators and block executors in HPX. These allocators and executors are NUMA-aware and minimize cross NUMA traffic by utilizing Linux's first touch policy for better data-locality. Furthermore, the block allocators allocate aligned memory, which is amenable for performance in vectorization of code.

```
1hpx::for_each(policy, begin, end,
2    [](auto& t)
3    {
4        auto& x = hpx::get<0>(t);
5        auto y = hpx::get<1>(t);
6        x = 5 * x + y;
7    });
```

Listing 2: **for_each** algorithm with memory-bound kernel.

Listing 2 lists a kernel that performs a saxpy operation. This saxpy operation has the low Arithmetic Intensity of 1/12 FLOPs/Bytes for floats. Therefore, the kernel can be found in the memory-bound section of the roofline model. Additional vectorization may only saturate the memory-bandwidth further, leading to a minor gain in performance over scalar execution. As observed on Line 4 and Line 5, we use a special iterator object called **zip_iterator** to pack multiple arguments into a single scalar or vector type.

```
1hpx::transform(policy, nums.begin(), nums.end(),
2    nums2.begin(), nums3.begin(),
3    [](auto x, auto y)
4    {
5        for (int i = 0; i < 100; i++)
6            x = 5 * sin(x) + 6 * cos(y);
7        return x;
8    });
```

Listing 3: **transform** algorithm with compute-bound kernel.

```
1hpx::transform(policy, nums.begin(), nums.end(),
2    nums2.begin(), nums3.begin(),
3    [](auto x, auto y)
4    {
5        return 5 * x + y;
6    });
```

Listing 4: **transform** algorithm with memory-bound kernel.

Listing 3 and Listing 4 describe the **transform** algorithm with compute-bound and memory-bound kernel respectively. The code explanation can be extrapolated from the description provided with Listings 1 and 2.

```
1hpx::count(policy, nums.begin(), nums.end(), gen());
2hpx::find(policy.on(executor), nums.begin(), nums.
    end(), 0);
```

Listing 5: **count** and **find** algorithm invocation

Listing 5 describes the second class of algorithms, i.e., the algorithms based on SIMD reductions. The **count** algorithm counts the number of occurrences of a random number produced using the **mersenne_twister_engine** generator[2]. The **find** algorithm traverses the container to find the first occurrence of the value specified, 0 in the example above.

Across all listings, we can notice that all the algorithms take a lambda with generic types as an argument. The iterator of the container must also be a random access iterator or our **simd** and **par_simd** implementations would fall back to **seq** and **par** respectively. In consequence:

1) Using SIMD policies in user code is trivial on two conditions: the functions utilized in the lambda need to have vectorized implementations and the iterator over the container has to be random access.
2) By bringing minimal changes to the code, significant speed-ups can be observed for compute-bound kernels and minor speed-ups can be observed for memory-bound kernels.

B. Mandelbrot

We ran the benchmarks over the complex plane from $x = -2.0$ to $x = 1.0$ and $y = -1.0$ to $y = 1.0$ over 4096 points in both x-axis and y-axis with a maximum of 2000 iterations at each point.

```
1template <typename ExecutionPolicy>
2auto mandel(ExecutionPolicy&& policy,
3            std::string const& fname,
4            float x_min, float x_max, float y_min,
5            float y_max, unsigned int width,
6            unsigned int height, float iter)
7{
8    auto seq_or_par_policy = get_base_policy<
         ExecutionPolicy>::value;
9    auto seq_or_simd_policy = get_simd_policy<
         ExecutionPolicy>::value;
10
11   /* <Initialization code...> */
12
13   hpx::for_loop(seq_or_par_policy, 0, height,
14   [=, &image](int i)
15   {
16       auto width_begin = image.begin() + width*i;
17       hpx::for_each(seq_or_simd_policy,
18           width_begin, width_begin + width,
19       [=, &image](auto &j)
20       {
21           using Vect = std::decay_t<decltype(j)>;
22           using Mask = get_mask_type_t<Vect>;
23
24           Vect x = x_min + (j) * dx;
25           Vect y = y_min + (i) * dy;
26           Vect zr = x, zi = y;
27           Vect count(0); Mask msk(0);
28           for (float k = 0; k < iterations; k++)
29           {
30               Vect r2 = zr * zr;
31               Vect i2 = zi * zi;
32               Mask currmsk = (r2 + i2) > 4;
33               if (all_of_simd(currmsk))
34               {
35                   mask_assign(
36                       msk ^ currmsk, count, k);
37                   break;
38               }
39               mask_assign(
40                   msk ^ currmsk, count, k);
41               msk = currmsk;
42               zi = 2 * zr * zi + y;
43               zr = r2 - i2 + x;
44           }
```

[2]https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine

```
45            count /= iterations;
46            count *= 255;
47            j = count;
48        });
49    });
50}
```

Listing 6: A uniform Mandelbrot implementation for all execution policies.

In Listing 6, we implement the Mandelbrot algorithm, which takes as arguments: an execution policy, x and y as limits of the complex plane, and the maximum number of steps for computation at a single point of the Mandelbrot equation.

We perform the benchmark with 4 policies namely **seq**, **par** , **simd** and **par_simd** and compare their speed-ups against the sequential policy. The **seq_or_par_policy** is a non-vectorized execution policy, i.e., either the sequential or the parallel policy whereas the **seq_or_simd_policy** is a scalar sequential or vectorized sequential **simd** policy. We perform vectorization over the inner-loop due to the benefits received from vectorizing contiguous elements. On Line 21, **Vect** is a data-parallel type for SIMD policies and a scalar type for others. We use the meta-template **get_mask_type** to fetch the type being used in conditional expressions. This **Mask** type is a **bool** when **Vect** is a scalar and of SIMD mask type when **Vect** is a data-parallel type. We handle the conditional statements using two utility functions, namely **all_of_simd** and **mask_assign** used on Line 33 and 35, respectively. This helps us in maintaining a single implementation to work with different execution policies with basic arithmetic data types and data-parallel types.

As observed in the inner loop, the Arithmetic Intensity of Mandelbrot set computation is very high as operations are carried on the same set of registers that is later stored into memory. Therefore, this implementation is compute-bound and amenable to significant speed-ups using data-parallel types.

V. System Setup

We run our benchmarks on four different processors supporting four different vector extensions. These systems are described in details in Appendix A. The benchmarks are therefore run on 8 float-wide AVX2 vector-registers (AMD EPYC 7H12), 16 float-wide AVX512 vector-registers (Intel Xeon Platinum 8260) and 4 float-wide NEON128 vector-registers (HiSilicon Kunpeng 916/920).

The detailed list of HPX's dependencies and versions used for the benchmarks is presented in Appendix A, along with the tags of the repositories created for storing the benchmarks code.

As we write the paper, only GCC 11.1 and Clang 12 have implementations of SIMD in the **std::experimental** namespace. We focus on GCC 11.1 for benchmarking for its better support. We have made all the benchmarks publicly available in the std-simd-perf[3] repository on GitHub. The repository consists of various branches, each representing results for a unique architecture. We use the compiler flag `-fno-tree-vectorize` for artificial codes, effectively disabling auto-vectorization. The artificial codes are trivially auto-vectorizable and do not represent real-world scenarios. Therefore, to better contrast with speed-ups relative to the sequential policy, we disable any auto-vectorization from the compiler end, allowing us to better understand the effects of explicit vectorization. However, we do not disable auto-vectorization for our real-world example, Mandelbrot. This allows us to test the actual performance improvements one can expect from a fairly complex conditional loop statement that may be hard to auto-vectorize. We pass the architecture flag `-march=native` to get the largest vector-register available on the target architecture as well as to simulate a portable code. For all the benchmarks, we utilize HPX's block allocators which are NUMA aware. It ensures that the memory is aligned, effectively improving the performance by reducing miss-aligns for vector load-store operations. All artificial benchmarks were run 5 times, and the average execution time is considered to report speed-ups against sequential execution. We run the Mandelbrot benchmark for a total of 10 times, and similarly the average execution time is considered to report speed-ups against sequential execution. All the plots are generated using Python's matplotlib[4] library.

VI. Results

A. Artificial Benchmarks

Figure 3 describes the results obtained from basic math operations available in the ISO C++ math header. All functions shown in Figure 3 support both basic scalar and data-parallel types. The implementation of the data-parallel types can be found in the **std::experimental** namespace. The y-axis is indicating the speed-up obtained with the SIMD policy compared to the sequential policy, i.e., the ratio of the execution time using the **seq** policy on the execution time using the **simd** execution policy.

We chose **sin** and **cos** as compute-bound kernels in artificial benchmarks as they are computationally more intensive. Furthermore, they are known to be efficiently ported to data-parallel types. This can be observed in the Figure 3 as enabling SIMD offers significant speed-ups compared to sequential execution times. We observe similar results when replacing floats with doubles. The benchmarks were performed on AMD EPYC 7H12 utilizing AVX2 vector-registers that are 256 bit wide and packs either 8 floats or 4 doubles. Note that the speed-ups for **sin** and **cos** go over the vector lane width 8 for floats or 4 for doubles. This is caused by varying scalar and vector implementations of these functions. Vector implementations (GCC) utilize both fewer instructions (fewer than it would be on explicit vectorization) and vector instructions. Therefore, we observe a super-linear scaling behavior. We do not observe any speed-ups in case of the other math

[3]https://github.com/srinivasyadav18/std-simd-perf/tree/master_v6

[4]https://matplotlib.org/

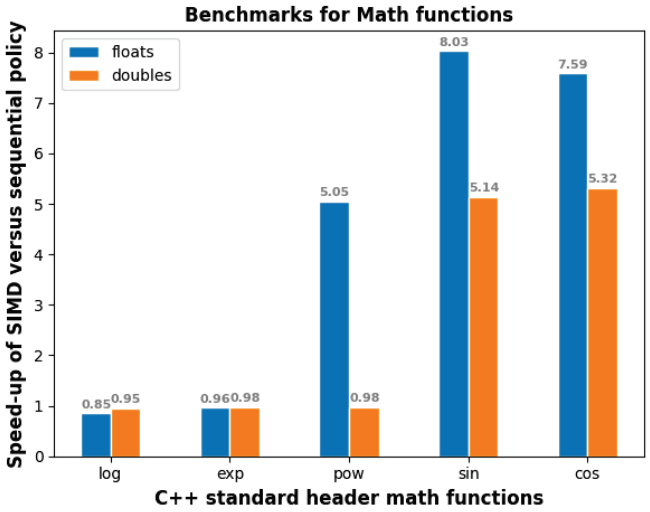

Fig. 3: Speed-ups observed in execution times of vectorized functions under C++ Math header relative to sequential code execution on AMD EPYC 7H12 with AVX2.

functions **log** and **exp** since they currently fall back to scalar implementation.

Algorithm	seq	simd	par	par_simd	AI
transform compute_bound	0.66	6.85	80.82	663.17	166.7
transform memory_bound	1.49	3.15	10.08	10.12	0.083
for_each compute_bound	0.66	5.07	72.32	528.43	250
for_each memory_bound	2.1	4.39	14.4	13.09	0.083
count	2.26	6.34	27.31	23.92	0.25
find	1.54	2.37	14.19	3.19	0.125

TABLE I: GFLOPS/s for artificial codes benchmarks with floats on AMD EPYC 7H12 with AVX2 with AI being Arithmetic Intensity in FLOPs/Bytes

Figure 4 depicts the benchmarks representing the speed-up obtained with the specified policy compared to the sequential execution times for different artificial codes with compute-bound and memory-bound kernels using floats. We also performed the benchmarks for doubles and the results are very similar and therefore not presented in this paper. We compare the speed-ups of three different execution policies, namely **simd**, **par**, and **par_simd**, against sequential execution times. We observe that the compute-bound algorithms outperform the memory-bound kernels with significant difference in speed-ups. This is expected as memory-bound kernels are limited by memory-bandwidth and scale up as long as the memory-bandwidth is not fully saturated. The compute-bound kernel performs sine and cosine operations for a total of 2000 arithmetic operations for floats and 4000 for doubles. This allows the compute-bound kernel to utilize as much of the available compute power as possible. Note, we observe speed-ups that go over the vector-length because of super-linear scaling in compute-bound. This can be explained by simply extrapolating the results obtained from Figure 3. We get similar benefits both when switching from **par** to **par_simd** and when switching from **seq** to the **simd** policy. We get 8.22x and 7.33x increase in speed-up from **par** to **par_simd** respectively for **transform** compute-bound and **for_each** compute-bound. The memory-bound algorithms shows expectedly low or no speed-up, as optimizing computations does not improve what is originally a memory bottleneck. It can even degrade the performance because of overheads caused by vector load and store operations.

To determine the system memory (DRAM) bandwidth, we used the Triad kernel of the STREAM benchmark[5] on arrays of 1 billion elements. We obtained 25.2 GB/s for single core and 140GB/s for all cores on AMD EPYC 7H12 (128 cores). In Figure 5, we visually confirm that **count** and **transform** memory bound have low Arithmetic Intensities. This characteristic is increased by the availability of the FMA instructions on the chosen architecture which reduces the AI by half for memory bound benchmarks (considering our saxpy kernel). We can observe only a small computational performance improvement from the sequential to the **simd** policy along with no improvement when switching from the parallel to the **par_simd** policy. This results are coherent since those benchmarks present a memory bottleneck. Note that the performance for the **simd** policy of **transform** memory bound is going over the memory bandwidth limit. This is due to the use of FMA instructions and due to several cache effects as this Roofline model is computed for DRAM.

On the other side of Figure 5, we can see an important increase in computational performance for both **for_each** compute bound and **transform** compute bound, which is expected for compute bound kernels. We can also notice that the compute bound benchmarks show room for improvements to reach the best performance. But this would require an investment in optimization from the user and those optimizations might also not be portable across architectures. This paper focuses on speed-ups obtained with minimal efforts from the user.

B. Mandelbrot

We computed the Mandelbrot set on four different machines with different architectures and vector lengths. The vector-length ranges from 4 floats-wide with NEON128 to 16 floats-wide with AVX512. Figure 6 describes the speed-ups relative to sequential execution with three execution policies, namely **simd**, **par**, **par_simd**. We enable auto vectorization for this benchmark to compare the performance of explicit vectorization against compiler's auto-vectorization.

We ported the HPX source code to an ISO C++20 compliant code to utilize the **unseq** and **par_unseq** policies but failed to produce consistent results. The **par_unseq** implementation within GCC 11.1 relies on Intel's Thread Building Blocks [25] to efficiently parallelize the standard algorithms. Unfortunately, we experienced no speed-ups utilizing the parallel policy with Intel TBB on certain machines (both x86 and Arm). In consequence, we are not reporting those results. Furthermore, utilizing the **unseq** policy does not speed-up our Mandelbrot code or any other similar code that relies

[5]https://www.cs.virginia.edu/stream

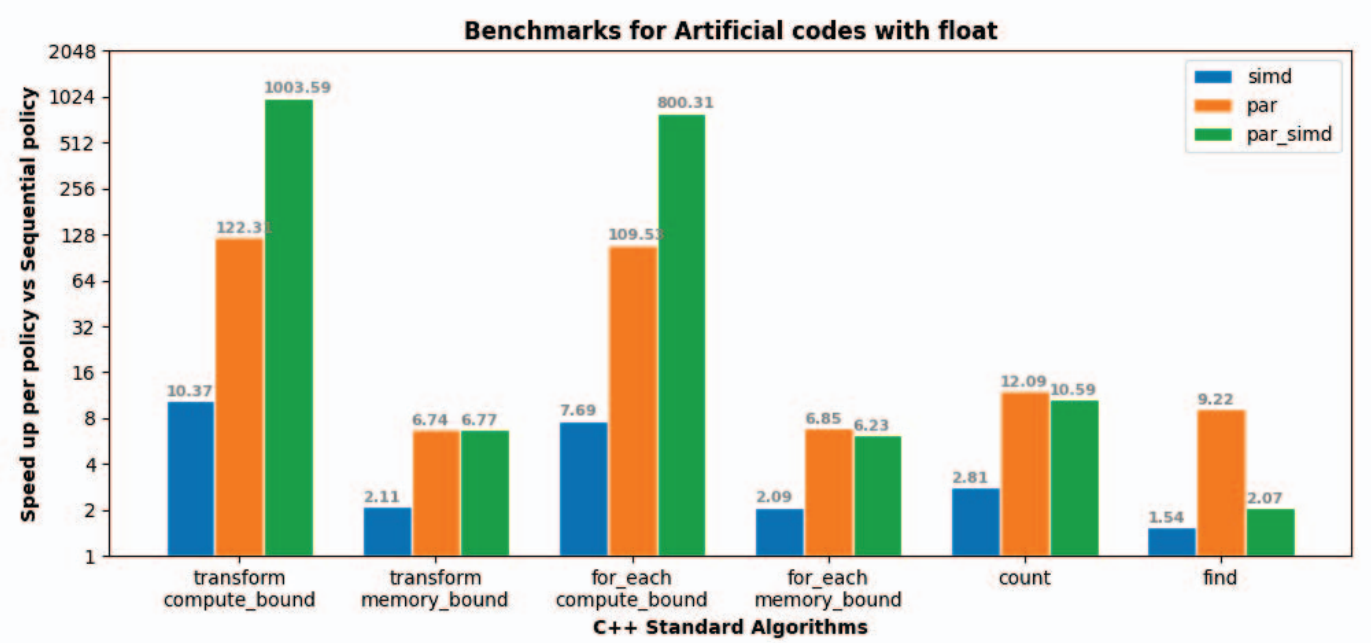

Fig. 4: Speed-ups of artificial codes with **`simd`**, **`par`**, **`par_simd`** policies relative to **`seq`** policy on AMD EPYC 7H12 with AVX2 and a vector pack of 8 floats

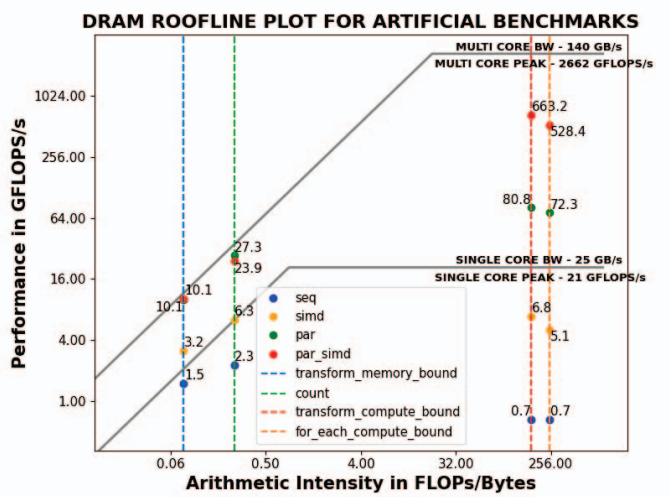

Fig. 5: Roofline model applied to Artificial code benchmarks performed on AMD EPYC 7H12 with AVX2 using floats

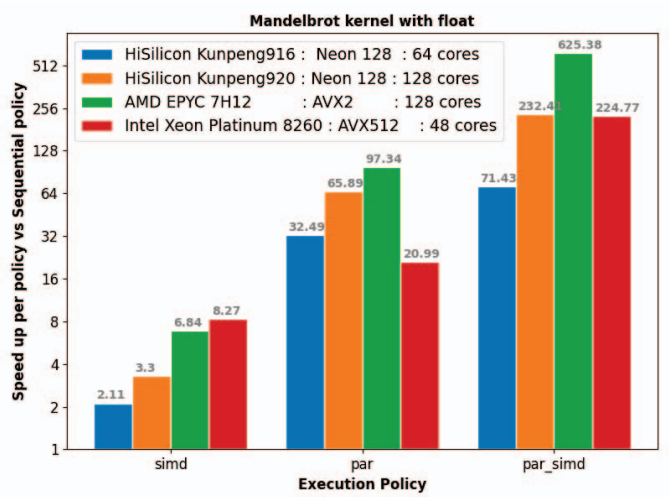

Fig. 6: Benchmarks for different execution policies of the Mandelbrot set. The vector pack sizes used for SIMD are 4 floats for HiSilicon, 8 for AMD, and 16 for Intel.

on control flow statements due to the limitations of OpenMP SIMD directive [26]. The behavior of **`unseq`** and **`seq`** showed no speed-up and is therefore not reported in this paper.

On Figure 6, we can see minimum two-orders of magnitude higher speed-ups over corresponding non-SIMD execution policies for all architectures. This is expected due to the compute-bound nature of the Mandelbrot set and its amenability towards instruction-level and thread-level parallelism. We see that the performance saturates at approximately 8.27x speed-up over sequential when using the **`simd`** policy relying on AVX512 vector intrinsics. The AMD architecture used performs particularly well since the computation of the Mandelbrot set using the SIMD policy is reaching 85.5% of the theoretical peak performance using a vector pack of 8 floats. Similarly, we obtain a high speed-up using the **`par_simd`** execution policy reaching 6.42x the speed-up of the **`par`** policy on AVX2.

Those previously described speed-ups are reached when the compiler auto-vectorization is enabled, it allows us to conclude that auto-vectorization is not sufficient on itself, and we benefit from explicit vectorization for this compute-bound problem.

VII. Conclusions

In this paper, we report on our experiences and performance analysis results for our implementation of two data-

parallel execution policies usable with HPX's parallel algorithms' module (**simd** and **par_simd**). We utilize the new experimental implementation of data-parallel types provided by recent versions of the GCC and Clang C++ standard libraries. The benchmark results collected from artificial tests and real-world codes presented in this paper are very promising. Compared to sequenced execution, we report on speedups of more than three orders of magnitude when executed using the newly implemented data-parallel execution policy **par_simd** with HPX's parallel algorithms. We also report that our implementation is performance portable across different compute architectures (x64 – Intel and AMD, and Arm), using different vectorization extensions (AVX2, AVX512, and NEON128). This implementation allows us to benefit lazily in the future from hardware improvements on vector registers (sizes, number).

VIII. Future Work

HPX currently lacks support for **unseq** and **par_unseq** execution policies. The natural next step would be to implement and integrate these policies into HPX. We acknowledge that the performance of these hint-based policies are highly dependent on the compiler. But trivial conversion of a non-vectorized policy requires all sub-functions to have vector implementations. While there is an ongoing effort to have vectorized implementations in both GCC and Clang, this may not be the case for libraries that are yet to be vectorized. We plan on adapting more algorithms to using the vectorization policies implemented in HPX. Furthermore, optimizations to the current implementation of vector execution policies by supporting different data-localities and strides will lead to a more generalized approach to implementing vector policies.

IX. Acknowledgements

We would like to acknowledge the support we received in the context of the Google Summer of Code project "Add vectorization to **par_simd** implementations of Parallel Algorithms". Furthermore, we acknowledge the ongoing generous support from the Center of Computation and Technology at Louisiana State University and the Swiss National Supercomputing Centre. Finally, we are thankful to Prof. Dirk Pleiter at Julich Supercomputing Center, Germany for providing us access to the Juawei cluster for performance benchmarking on Arm based systems.

References

[1] The C++ Standards Committee, "ISO International Standard ISO/IEC 14882:2017, Programming Language C++," Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep., 2017, http://www.open-std.org/jtc1/sc22/wg21.

[2] ——, "ISO International Standard ISO/IEC 14882:2020, Programming Language C++," Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep., 2020, http://www.open-std.org/jtc1/sc22/wg21.

[3] "N4577: Working Draft: Technical Specification for C++ Parallelism Version 2," Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep., 2018, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4755.pdf.

[4] "The GNU C++ Library," https://gcc.gnu.org/onlinedocs/libstdc++/, 2021.

[5] "libc++ 13.0 documentation," https://libcxx.llvm.org/, 2021.

[6] "P0350R1: Integrating simd with parallel algorithms," Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep., 2020, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0350r4.pdf.

[7] H. Kaiser, B. Adelstein-Lelbach, T. Heller, and A. B. et.al., "HPX V1.7: The C++ Standard Library for Parallelism and Concurrency," 2021, http://dx.doi.org/10.5281/zenodo.598202.

[8] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, "HPX - The C++ Standard Library for Parallelism and Concurrency," vol. 5, no. 53, p. 2352. [Online]. Available: https://doi.org/10.21105/joss.02352

[9] "Boost Software License V1," https://www.boost.org/users/license.html, 2021.

[10] D. Nuzman and A. Zaks, "Autovectorization in GCC—two years later," *Proceedings of the 2006 GCC Developers Summit*, 06 2006.

[11] J. Huber, W. Wei, G. Georgakoudis, J. Doerfert, and O. R. Hernandez, "A Case Study of LLVM-Based Analysis for Optimizing SIMD Code Generation," *CoRR*, vol. abs/2106.14332, 2021. [Online]. Available: https://arxiv.org/abs/2106.14332

[12] W. Killian, R. Miceli, E. Park, M. A. Vega, and J. Cavazos, "Partnership for Advanced Computing in Europe Performance Improvement in Kernels by Guiding Compiler Auto-Vectorization Heuristics," 2014.

[13] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, "Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 265–274. [Online]. Available: https://doi.org/10.1145/1995896.1995938

[14] C. Newburn, F. Wende, M. Noack, T. Steinke, M. Klemm, and G. Zitzlsberger, "Portable SIMD Performance with OpenMP 4.x Compiler Directives," 08 2016.

[15] J. N. Huber, O. R. Hernandez, and M. G. Lopez, "Effective Vectorization with OpenMP 4.5," 2017.

[16] B. Ren, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, "Exploiting Vector and Multicore Parallelism for Recursive, Data- and Task-Parallel Programs," *ACM SIGPLAN Notices*, vol. 52, pp. 117–130, 01 2017.

[17] A. Al Hasib, L. Natvig, P. Kjeldsberg, and J. Cebrian, "Energy Efficiency Effects of Vectorization in Data Reuse Transformations for Many-Core Processors - A Case Study," *Journal of Low Power Electronics and Applications*, vol. 7, 02 2017.

[18] J. Lapresté, J. Falcou, P. Esterie, and M. Gaunard, "Exploiting multimedia extensions in c++: A portable approach," *Computing in Science & Engineering*, vol. 14, no. 05, pp. 72–77, sep 2012.

[19] M. Kretz and V. Lindenstruth, "Vc: A C++ library for explicit vectorization," *Software: Practice and Experience*, vol. 42, no. 11, pp. 1409–1430. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1149

[20] R. Leißa, I. Haffner, and S. Hack, "Sierra: a SIMD extension for C++," in *WPMVP '14*, 2014.

[21] G. Amadio, P. Canal, D. Piparo, and S. Wenzel, "Speeding up software with VecCore," *Journal of Physics: Conference Series*, vol. 1085, p. 032034, sep 2018. [Online]. Available: https://doi.org/10.1088/1742-6596/1085/3/032034

[22] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[23] S. Williams, D. Patterson, L. Oliker, J. Shalf, and K. Yelick, "The roofline model: A pedagogical tool for auto-tuning kernels on multicore architectures," in *Hot Chips*, vol. 20, 2008, pp. 24–26.

[24] S. W. Williams, *Auto-tuning performance on multicore computers*. University of California, Berkeley, 2008.

[25] A. Kukanov and M. J. Voss, "The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[26] J. Huber, O. Hernandez, and G. Lopez, "Effective vectorization with openmp 4.5," *ORNL/TM-2016/391. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States). Oak Ridge Leadership Computing Facility (OLCF)*, 2017.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: PARALLEL SIMD - A POLICY BASED SOLUTION FOR FREESPEED-UP USING C++ DATA-PARALLEL TYPES

A. Abstract

The appendix contains information to build the benchmarks and their dependencies, along with instructions used to execute the binaries to generate the results as provided in Section 6 of the paper. We provide the compilers and libraries used to compile and run HPX and std-simd-perf. In addition, we provide python scripts to generate performance plots used in the paper.

B. Description

1) Check-list (artifact meta information):

- **Algorithms:** Artifiial Codes, Mandelbrot set computation.
- **Program:** All binaries are generated through the compilation of CMake targets.
- **Compilation:** GCC 11.1.0, CMake 3.19.1
- **Hardware:**
  1) Rostam cluster, LSU, USA: The Rostam mi100 cluster is equipped with two 64-core AMD EPYC 7H12 CPUs coupled with 512 GB of memory. The AMD processor supports up to AVX2, and the benchmarks are run on 8 float-wide AVX2 vector-registers.
  2) Loni cluster, LSU, USA: The Loni QBC3 cluster is equipped with two 24-core Intel Cascade Lake (Intel® Xeon® Platinum 8260 Processor) CPUs coupled with 192 GB of memory. The Intel Processor supports up to AVX512, and the benchmarks are run on 16 float-wide AVX512 vector-registers.
  3) Juawei cluster, Julich Supercomputing Center, Germany: The Juawei cluster is equipped with two 64-core HiSilicon Kunpeng 920 CPUs coupled with 256 GB of memory. Furthermore, we also utilize the two 32-core HiSilicon Kunpeng 916 CPUs coupled with 256 GB of memory. The benchmarks were run on 4 float-wide NEON128 vector-registers on Kunpeng 916 and Kunpeng 920.
- **Execution:** Jobs were submitted to the nodes using slurm.
- **Output:** A csv file containing the size of the array, the number of threads used and the execution times for different policies.
- **Experiment workflow:** Variations in input sizes.
- **Publicly available?:** Yes

2) How software can be obtained (if available): HPX (ec871f2)[1], and std-simd-perf (6008996)[2] can be obtained via GitHub.

3) Hardware dependencies: Rostam mi100, Loni QBC3, Juawei2 cluster.

4) Software dependencies: For this paper, the following compilers and libraries with their corresponding versions were utilized:

Package Name	Version (or Commit Hash)
GCC	11.1
hwloc	2.4.1
TCMalloc	2.7
Boost	1.76.0
CMake	3.19.5
std-simd-perf	6008996
HPX	ec871f2
Python	3.9.2

TABLE I
LIST OF DEPENDENCIES OF HPX AND ADDITIONAL BENCHMARKS

C. Installation

We assume that the prerequisites are already installed on the system at their default locations, thus relieving us from adding specific paths to CMake. We assume that Boost was compiled with the same compiler, which was used to compile HPX.

[1]https://github.com/srinivasyadav18/hpx/tree/simd_algorithms_v3

[2] https://github.com/srinivasyadav18/std-simd-perf/tree/master_v6

Listing 1. Compile and install HPX

```
$ module load gcc/11.1 hwloc boost
$ git clone --single-branch -b simd_algorithms_v3 \
   https://github.com/srinivasyadav18/hpx.git
$ cd hpx && mkdir build && cd build
$ cmake \
   -DCMAKE_BUILD_TYPE=Release \
   -DHPX_WITH_CXX20=ON \
   -DHPX_WITH_FETCH_ASIO=ON \
   -DCMAKE_INSTALL_PREFIX=<path_to_hpx> \
   ..
$ make -j<n_cores>
$ make install
```

Listing 2. Compile std-simd-perf

```
$ module load gcc/11.1 hwloc boost
$ git clone --single-branch -b master_v6 \
   https://github.com/srinivasyadav18/std-simd-perf.git
$ cd std-simd-perf && mkdir build && cd build
$ cmake -DCMAKE_BUILD_TYPE=Release \
   -DHPX_DIR=<path_to_hpx> \
   -DSIMD_CORES=<n_cores> \
   ..
$ make -j<n_cores>
```

D. Experiment workflow

All the benchmarks reside in the std-simd-perf[2] Github repository using CMake Build System which for building and runnning the tests (through `ctest`).

The results are obtained with the average of 5 runs for artificial codes and the average of 10 runs for the mandelbrot kernel.

E. Evaluation and expected result

For the artificial codes benchmarks, we implemented the kernels in C++ with both explicit vectorization and disabling the auto vectorization. We evaluated the performance by running the benchmarks with different execution policies and we compared the different performance speed-ups of those policies against the sequential policy. Each test ran with `ctest` generates a csv file for the type used. Listing 3 shows an example file containing the output for all four execution policies (`seq`, `simd`, `par`, `par_simd`). Note that in Listing 3, `n` denotes the number of elements in the array in `log2` scale.

Listing 3. Example output of Artificial code - transform compute bound kernel with float type on AMD EPYC

```
$ cat transform_compute_bound/plots/float.csv
n,lane,threads,seq,simd,par,par_simd
27,8,128,0.179641,0.085323,0.0266435,0.026521
```

For the Mandelbrot benchmark, we implemented the kernel in C++ and evaluated the performance through execution times from different execution policies. Listing 4 shows an example file containing the output of executing mandelbrot kernel.

Listing 4. Example output of mandelbrot kernel benchmark

```
$ cat kernels/mandel/plots/float.csv
n,lane,threads,seq,simd,par,par_simd,unseq,par_unseq
16777216,8,128,25.7844,3.76265,0.263635,0.0421223,
25.7734, 0.26903
```