SN Computer Science (2024) 5:459
https://doi.org/10.1007/s42979-024-02769-6

ORIGINAL RESEARCH

Shared Memory Parallelism in Modern C++ and HPX

Patrick Diehl[1,2] · **Steven R. Brandt**[1] · **Hartmut Kaiser**[1]

Received: 25 July 2023 / Accepted: 4 March 2024

Abstract
Parallel programming remains a daunting challenge, from struggling to express a parallel algorithm without cluttering the underlying synchronous logic to describing which tools to employ to ensure a calculation is performed correctly. Over the years, numerous solutions have arisen, requiring new programming languages, extensions to programming languages, or adding pragmas. Support for these various tools and extensions is available to varying degrees. In recent years, the C++ standards committee has worked to refine the language features and libraries needed to support parallel programming on a single computational node. Eventually, all major vendors and compilers will provide robust and performant implementations of these standards. Until then, the HPX library and runtime provide cutting-edge implementations of the standards and proposed standards and extensions. Because of these advances, it is now possible to write high performance parallel code without custom extensions to C++. We provide an overview of modern parallel programming in C++, describing the language and library features and providing brief examples of how to use them.

Keywords C++ · HPX · AMT · Parallelism

Introduction

Parallel programming is essential to modern software development and is supported in recent programming languages like Julia or Rust. However, in older languages such as C++, parallel programming features were not originally included as language or library features.

To address this omission, POSIX threads [1], so-called *pthreads*, a C library, was created for the Unix operating system. The application program interface (API) for *pthreads* was defined by the *POSIX.1C* thread extension (*IEEE Std 1003.1c-1995*). Likewise, with the C++ 11 standard [2], `std::thread` was added in C++ as a low level interface. At a higher abstraction layer, `std::async` and `std::future` for asynchronous programming were added.

In addition, the standard added utilities to make parallel programming easier (e.g. smart pointers and lambda functions). With the C++ 14 standard [3], these utilities were further augmented with generic lambda functions and shared mutexes.

To make parallel programming more accessible and less error-prone, the C++ 17 standard [4] introduced parallel algorithms, allowing programmers to execute most of the algorithms from the C++ 98 standard in parallel (e.g. `std::sort` or `std::reduce`).

Coroutines were added with the C++ 23 standard to support asynchronous programming. The keywords `co_return`, `co_yield`, and `co_await` added the ability to suspend and resume functions. Also, in the C++ 23 standard, the *ranges* library was added, which can be seen as the generalization and extension of the algorithm library. Finally, utilities such as semaphores, latches, and barriers were added. Soon, it is expected that `std::async` will become deprecated to be succeeded by the sender and receiver library (which has yet to be accepted). Figure 1 shows the timeline

This article is part of the topical collection "Applications and Frameworks using the Asynchronous Many Task Paradigm" guest edited by Patrick Diehl, Hartmut Kaiser, Peter Thoman, Steven R. Brandt and Ram Ramanujam.

✉ Patrick Diehl
patrickdiehl@lsu.edu

Steven R. Brandt
sbrandt@cct.lsu.edu

Hartmut Kaiser
hkaiser@cct.lsu.edu

[1] Center of Computation and Technology, Louisiana State University, Digital Media Center, Baton Rouge, LA 70803, USA

[2] Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA 70803, USA

Published online: 20 April 2024

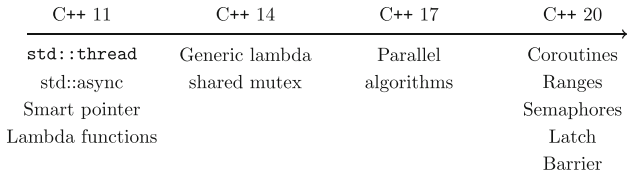

Fig. 1 Timeline of the parallel features added to the C++ standard from C++ 11 to C++ 20

of parallel features added to the C++ standard from C++ 11 to C++ 20.

The C++ Standard Library for Parallelism and Concurrency (HPX) implements all the latest parallel programming features, both proposed and accepted in the C++ standard. In addition, HPX provides extensions to the functionality of the standard, providing mechanisms for distributed parallel programming, alternative ways to create asynchrony, and more.

What is HPX? HPX is an asynchronous many-task runtime system. HPX employs light-weight (user-level) threads that are cooperatively scheduled on top of operating system threads and performs context switches to enable blocked threads to get back to work.

For more details about HPX, we refer to Section "HPX". Because HPX conforms to the C++ standard, any conforming C++ code can be easily converted to HPX by changing some headers and namespaces. To conclude, while single node parallelism is included in the C++ standard and no external libraries or language extensions are needed, HPX provides a reliable way to stay on the cutting edge of the standard.

In this paper, we will introduce asynchronous programming, parallel algorithms, and coroutines, senders and receivers (see P2300), and compare the performance between (standard) C++ using operating system threads and HPX using light-weight threads. Finally, we will discuss the benefits of each approach.

The paper is structured as follows: Section "Related Work" gives a brief overview of related work. Section "HPX" introduces HPX and the features described in this paper. In Section "Approaches", four approaches to implementing the Taylor series of the natural logarithm are provided. Section "Comparison" compares the programming paradigms used in these approaches. Section "performance" compares the performance of the approaches on Intel, AMD, and A64 FX CPUs. Finally, Section "Conclusion" concludes the work.

Related Work

In the past, parallelism in C++ was usually achieved by using OpenMP [5] and Cilk [6] as language extensions. Alternatively, Intel Thread Building Blocks (TBB), Microsoft Parallel Patterns Library (PPL) provided access to parallelism through libraries. More recently, Kokkos [7] has provided a library interface for parallel and heterogeneous computing. While all these approaches have different advantages, they also have different interfaces, and none are part of C++ standard. Conforming to the standard might mean that future versions of a conforming code compile and run more reliably, and this is a critical consideration among many in constructing a new parallel program or adding parallelism to an existing code.

Table 1 Availability of the approaches studied in other run time systems. However, HPX alone is studied in this paper. Therefore, HPX is the basis for the comparison of the features. With ~ we indicated that the features are only partially supported

Approach	Futurization	Coroutines	Parallel Algorithms	Sender and Receivers
HPX	✓	✓	✓	✓
Charm++	✓	~	X	X
Chapel	✓	X	~	X
UPC++	✓	X	X	X

Another longtime player in the asynchronous many-thread library arena is Charm++ [8]. Like HPX, Charm++ also provides facilities for distributed programming (for which, at present, the C++ provides no standard). For a comparison of Charm++ and HPX with OpenMP and MPI (a widely accepted standard for distributed parallel programming) using Task Bench, we refer to [9]. Other notable Asynchronous Many Task Systems (AMTs) are: Chapel [10], X10 [11], and UPC++ [12]. For a more detailed comparison of AMTs, we refer to [13]. Table 1 lists the support of approaches, namely, futures (Section Futures and Futurization), coroutines (Section Coroutines), parallel algorithms (Section Parallel algorithms), and senders & receivers (Section Senders and Receivers) by other asynchronous many-task runtime systems (AMTs). Charm++ provides futures but not coroutines. It also provides functionality similar to senders and receivers. A *Chare* can be used somewhat like a scheduler, and a Charm++ callback can provide similar functionality to `then()`.

Chapel provides futures. Parallel algorithms are partially supported, e.g. parallel `for` loops. Coroutines and sender & receivers are not supported. UPC++ has futures but does not support the other features.

HPX

HPX [14] is an Asynchronous Many-task Runtime System (AMT) that exposes an ISO C++ standards conforming API for shared memory parallel programming, and extensions to that API library that enable distributed computing. This API enables asynchronous parallel programming through futures, senders and receivers, channels, and other synchronization

primitives. This API also eases the burden on a new programmer while learning to use HPX. It also makes it much easier to port code. HPX employs a user-level threading system that can fully exploit available parallel resources through fine-grain parallelism on various contemporary and emerging high-performance computing architectures. HPX makes it possible to create scalable parallel applications that expose excellent parallel efficiency and high resource utilization. HPX's asynchronous programming model enables intrinsic overlapping of computation and communication, prefers moving work to data over moving data to work, and does so while exposing minimal overheads.

In the context of this paper, we focus on assessing the performance of HPX's implementation of futures and parallel algorithms as mandated by the C++ 17, 20, and 23 standards.

HPX's Distributed Features

Looking at Figure 1, we see the parallel features recently added to the C++ standard. As of the latest C++ standard, no standard C++ primitives or features are available for distributed programming. Also, as far as the authors are aware, there is currently no discussion in the C++ standard committee to add distributed features. HPX is a pioneer in adding features for distribute programming. To do so, HPX provides the capability to launch remote functions and await their results with regular futures.

For internode communication, HPX uses a modular layer which can use either the Message Passing Interface (MPI), libabric [15], or LCI [16]. Recently, experimental support has been added for GASNet [17] and OpenSHMEM. However, this work is focused on the C++ primitives and features in the C++ standard for parallel programming. For distributed programming with HPX, we refer to [18]. Octo-Tiger [19], an astrophysics application for stellar mergers, is one example application using HPX's distributed features. Distributed runs have been studied on RISC-V single board computers [20] and supercomputers, e.g. Supercomputer Fugaku [21], Piz Daint [15], and Summit [22]. Since distributed programming is not in the C++ standard yet, this work focuses on the shared memory features which are in the current C++ 20 standard. Not that senders and receivers have not yet made it in the C++ standard, but are actively discussed.

Approaches

To showcase the various approaches to shared memory parallelism, we will first implement the Taylor series for the natural logarithm in parallel. The Maclaurin series for the natural logarithm ln with the basis e reads as

$$ln(1+x) = \sum_{n=1}^{\infty}(-1)^{n+1}\frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots, \text{with } |x| < 1. \tag{1}$$

Secondly, we will implement the Mandelbrot set [23], which is a set of complex values of $c \in \mathbb{C}$. The set is determined by iteratively solving the quadratic map

$$z_{n+1} = z_n^2 + c. \tag{2}$$

From the computer science perspective, the first Mandelbrot set was computed and visualized in 1978 [24]. Figure 2 shows a visualization of the Mandelbrot set using the code in the paper. The compute kernel of the Mandelbrot set is shown in Listing 1. The compute kernel has as its argument the complex number c which can be seen as the initial value of the Mandelbrot set. The Equation (2) is evaluated iteratively in the `for` loop for `max_iterations`. Is the absolute value of the complex number z is larger than two the pixel is marked by one which indicates that z is in the complex set. Is the absolute values is less than two the pixel is marked by zero which means that z is not in the complex set. All complex numbers not in the set are colored by black and all complex numbers in the set are transformed to red green blue values and colored accordingly. For simplicity, we use the **P**ortable **B**it**m**ap (PBM)[1] file format to store the images.

Listing 1 Computer kernel for the Mandelbrot set.

```
size_t compute_pixel(complex c) {
    std::complex<double> z(0, 0);

    for (size_t i = 0; i <
        max_iterations; i++) {
        z = z * z + c;
        if (abs(z) > 2.0) {
          return 1;
        }
    }
    return 0;
}
```

For simplicity, we will omit the main method and all headers from the code examples. However, we will mention the specific headers in the text, and we provide the complete code for all examples on GitHub®. The source for the Mandelbrot examples are in the appendix of the paper.

[1] https://netpbm.sourceforge.net/doc/pbm.html.

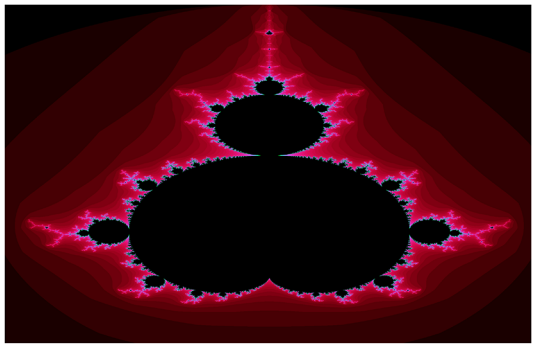

Fig. 2 Mandelbrot set computed with the code using 80 max iterations, `c = complex(0, 4)* complex(i, 0)/ complex (3840, 0)- complex(0, 2)` , and a size of 3840 × 2160 pixels

Futures and Futurization

The current abstractions for parallel programming in C++ are low-level threads `std::thread`, `std::async`, and `std::future`. However, in a future C++ standard, it is expected that some of these facilities will become deprecated and will be replaced by sender and receivers. HPX, however, will continue to support an extended version of futures which share many of the capabilities of senders and receivers, including a `then()` method, a `when_all()` method, executors, and so on.

Futures represent a proxy for a result that may not yet be computed and provide a relatively intuitive way to express asynchronous computations. The C++ standard allows programmers to retrieve the value of futures using the `.get()` method, but HPX allows programmers to attach a continuation to the future using the `then(std::function<T>)` method. This capability, combined with a `when_all()` method for waiting for future groups, makes it possible to write asynchronous subroutines and algorithms that never block. This is an essential consideration for libraries that rely on a pool of workers to carry out parallel computations. Blocking one or more of them might lead not only to slower code, but also blocked code. Routines that are rewritten in this way to run in parallel but without calling `.get()` are said to be *futurized*. As of this writing, futurized code is only possible with HPX, and not with the C++ standard.

Listing 2 shows the implementation of the Taylor series. The amount of work is divided equally among threads. In Line 14, a lambda function is launched to act on each chunk of work asynchronously and an `hpx::future<double>` is returned. Note that we do not need to wait for the lambda function to finish for the `for` loop to proceed. This happens because the `hpx::future` is a placeholder for the result of the lambda function, freeing us from the need to wait for it to be computed. In Line 29 a barrier is introduced to collect the partial results using `hpx::when_all`. Here, the HPX runtime waits until all futures are ready, which means that the computation in the lambda function has finished. In Line 30 we specify which lambda function is called. We use the `.get()` function to collect all the partial results. If the result is not ready, HPX would wait here for the result to be ready. However, when the callback provided to `hpx::when_all` is called, all results are ready. In Line 36, we need to call `.get()` since `hpx::when_all` returns a future for integration in the asynchronous dependency graph.

Listing 7 shows the implementation of the Mandelbrot set using futures. Again, the code structure is similar and we iterate over the rows of the image and asynchronously launch a function to compute the pixel color for a column of the image.

Coroutines

With C++ 20 coroutines, functions that can be suspended and resumed were added. The three following `return` types are available for coroutines: `co_return` which is similar to `return`, but the function is suspended; `co_yield` returns the expression to the caller and suspends the current coroutine; and `co_await` which suspends the coroutine and returns the control to the caller.

These new keywords can only be invoked inside routines that have special return types. The `hpx::future` class is one such type. It supports the use of `co_return` and `co_await` (but not `co_yield`, since `co_yield` is designed for generators which return a sequence of values and futures contain a single value).

When used with `hpx::future`, `co_await` is similar to `.get()` from the programmer's perspective. The difference is that `co_await` suspends a task rather than blocking. This means that it can more safely be run inside a group of worker threads. Note, however, that the HPX implementation of `.get()` can already suspend the task, so there is no semantic difference.

A coroutine version of Listing 2 can be found in Listing 3. In Line 5 of Listing 3, we define the function `run` as our coroutine by having it return an `hpx::future`. Next, we copy the code from Listing 2 for the evaluation of the Taylor series, however, we changed three lines to use the new coroutine features. First, in Line 33, we use `co_await` while we wait for all futures. Second, in Line 36, we use `co_await` to collect the partial results of all futures. Note in Listing 2, we had to call `.get()` here to wait for the futures. Third, in Line 36, we call `co_return` at the end of our coroutine. Note that internally HPX will call `.get()` where we use `co_await`, so the code is easier to read but will not run faster.

Listing 8 shows the implementation of the Mandelbrot set using futures + coroutines. Again, the code structure is similar and we iterate over the rows of the image and asyn-

Listing 2 Parallel implementation of the natural logarithm using `hpx::async` and `hpx::future`.

```
double run(size_t n, size_t num_threads, double x) {
  std::vector<double> parts(n);
  std::iota(parts.begin(), parts.end(), 1);

  size_t partition_size = n / num_threads;

  std::vector<hpx::future<double>> futures;
  for (size_t i = 0; i < num_threads; i++) {
    size_t begin = i * partition_size;
    size_t end = (i + 1) * partition_size;
    if (i == num_threads - 1) end = n;

    hpx::future<double> f = hpx::async(
        [begin, end, x, &parts]() -> double {
      std::for_each(parts.begin() + begin,
                    parts.begin() + end, [x](double& e) {
        e = std::pow(-1.0, e + 1) * std::pow(x, e) / (e);
      });

      return hpx::reduce(parts.begin() + begin,
                         parts.begin() + end, 0.);
    });

    futures.push_back(std::move(f));
  }

  double result = 0;

  hpx::when_all(futures)
      .then([&](auto&& f) {
        auto futures = f.get();

        for (size_t i = 0; i < futures.size(); i++)
            result += futures[i].get();
      })
      .get();

  return result;
}
```

chronously launch a function to compute the pixel color for a column of the image.

Parallel Algorithms

The algorithms within the C++ standard library introduced with the C++ 98 standard were extended with parallel execution in the C++ 17 standard. Listing 4 shows the complete code. In Line 15 we use the algorithm `std::for_each` to iterate over each element of the `std::vector` to evaluate the value x of the Taylor series. In Line 21 the algorithm `std::reduce` is used to compute the sum of all evaluations. Note that the only difference between the parallel version and the original C++ 98 standard is the first argument of both algorithms, the execution policy. The following execution policies in the header `#include <execution>` [25] are currently available:

- `std::execution::par`: The algorithm is executed in parallel using multiple operating system threads.
- `std::execution::seq`: The algorithm is executed in parallel using one operating system thread.
- `std::execution::par_unseq`: The algorithm is executed in parallel using multiple operating system threads and vectorization for additional optimizations.

Note that this is still an experimental feature and, as of this writing, only the GNU compiler collection (GCC) ≥ 9 and Microsoft Visual C++ compiler ≥ 15.7 support this feature. Intel's One API compiler uses Thread Building Blocks (TBB) to implement this feature.

The same functionality for `execution` of parallel algorithms is available within HPX.

Listing 9 shows the implementation of the Mandelbrot set in Equation (2) using HPX's parallel algorithms. Here, we

Listing 3 Example for the computation of the Taylor series for the natural logarithm using HPX's futures and coroutines.

```
#include <coroutine>

hpx::future<double> run(size_t n,
                        size_t num_threads,
                        double x) {
  std::vector<double> parts(n);
  std::iota(parts.begin(), parts.end(), 1);

  size_t partition_size = n / num_threads;

  std::vector<hpx::future<double>> futures;
  for (size_t i = 0; i < num_threads; i++) {
    size_t begin = i * partition_size;
    size_t end = (i + 1) * partition_size;
    if (i == num_threads - 1) end = n;

    hpx::future<double> f = hpx::async(
        [begin, end, x, &parts]() -> double {
      std::for_each(parts.begin() + begin,
                    parts.begin() + end, [x](double& e) {
        e = std::pow(-1.0, e + 1) * std::pow(x, e) / (e);
      });

      return hpx::reduce(parts.begin() + begin,
                         parts.begin() + end, 0.);
    });

    futures.push_back(std::move(f));
  }

  double result = 0;

  auto futures2 = co_await hpx::when_all(futures);

  for (size_t i = 0; i < futures2.size(); i++)
    result += co_await futures2[i];

  co_return result;
}
```

use `hpx::experimental::for_loop` to iterate over the range from `0` to `pixel_x-1` like with a regular `for` loop. However, this kind of parallel for loop is not yet in the C++ standard.

Additional HPX Features

However, HPX extends the current features available in the C++ 17 standard, allowing execution policies with chunk sizes to specify the amount of work each thread is operating on at once. The following chunk sizes are available:

- `hpx::execution::static_chunk_size`: The container elements are divided into pieces of a given size and then assigned to the threads.
- `hpx::execution::auto_chunk_size`: Chunk size is determined after 1% of the total container elements were executed.
- `hpx::execution::dynamic_chunk_size`: Dynamically scheduled among the threads and if one thread is done it gets dynamically assigned a new chunk.

For details about the effect of chunk sizes on performance, we refer to [26]. A machine learning approach to determining chunk size is presented here [27, 28]. With respect to vectorization, HPX provides the execution policy `hpx::execution::simd` to execute the algorithm using vectorization. In addition, HPX provides a combined execution policy `hpx::execution::par_simd` to combine parallelism and vectorization. Here, `std::experimental::simd` [29], Vc [30], and Eve are possible backends. Furthermore, HPX's parallel algorithms can be

Listing 4 Implementation of the Taylor series of the natural logarithm using C++ parallel algorithms.

```
#include <iostream>
#include <future>
#include <vector>
#include <algorithm>
#include <numeric>
#include <execution>
#include <cmath>

double run(size_t n, size_t num_threads, double x) {
    std::vector<double> parts(n);
    std::iota(parts.begin(), parts.end(), 1);

    std::for_each(std::execution::par,
                  parts.begin(),
                  parts.end(), [x](double& e) {
        e = std::pow(-1.0, e+1) * std::pow(x, e) / e;
    });

    double result = std::reduce(std::execution::par,
                                parts.begin(),
                                parts.end(), 0.);
    return result;
}
```

combined with asynchronous programming. Here, an `hpx::future` is returned and can be integrated into HPX's asynchronous execution graph.

Listing 5 shows the usage of the chunk size feature. In Line 4 a static chunk size of ten is defined and passed to the `hpx::for_each` in Line 9 by using `.with()`. In Line 12 the parallel algorithm `hpx::reduce` is wrapped into a future, which can be integrated within HPX's asynchronous dependency graph.

Senders and Receivers

A new framework for writing parallel codes is currently being debated by the C++ standards committee: senders and receivers. One of the goals of this framework is to make it easier to execute codes on heterogeneous devices. The various devices are expressed as *schedulers*. In principle, these could be GPUs, different NUMA domains, or arbitrary groups of cores.

Each step of a calculation is expressed as a *sender*. Senders are typically chained together using the pipe operator in analogy to the bash shell. Values, error conditions (or exceptions), as well as requests to stop a computation, can be carried through the pipeline.

By default, building the pipeline does nothing. Execution begins only when `ensure_started()`, `sync_wait()`, or `start_detached()` is called.

Receivers are usually implicit, hidden in the call to `sync_wait()` at the end.

We note that this proposal was not accepted into the C++ 23 standard, partly because it was proposed too close to the deadline. It may also need further development. In our experiments writing short codes to use senders and receivers, we attempted to write a recursive Fibonacci routine that took a sender as input and produced a sender as output and did not itself call `sync_wait()` to get the result. In order to write it, we needed to make use of the `any_sender<T>` class provided in the HPX implementation but not specified in the standard yet. Whether additions of this kind turn out to be necessary, or whether the proposal itself will ultimately be accepted remains for the committee to decide.

Listing 10 shows the implementation of the Mandelbrot set in Equation (2) using senders and receivers. The implementation is very similar. However, we do no need to use `.then` in Line 28 of Listing 6.

Comparison of the approaches in shared memory

In the previous section, the focus was on how to implement the Taylor series for the natural logarithm, see Equation (1), using the various approaches.

The fundamental difference in the approaches lies in where the various codes block and how much overhead they introduce. For the standard library, calls to `future.get()` will potentially block. In our parallel future Listing 2 we use `when_all()` which defers most of the calls to `get()` until

Listing 5 Implementation of the Taylor series of the natural logarithm using parallel algorithms.

```
#include <hpx/execution/executors/static_chunk_size.hpp>

double run(size_t n, size_t num_threads, double x) {
    hpx::execution::static_chunk_size scs(10);
    std::vector<double> parts(n);
    std::iota(parts.begin(), parts.end(), 1);
    hpx::for_each(
         hpx::execution::par.with(scs),
         parts.begin(), parts.end(),
         [x](double& e) { e = std::pow(-1.0, e+1) * std::pow(x, e) / e; });

    hpx::future<double> f =
         hpx::reduce(hpx::execution::par(hpx::execution::task),
                     parts.begin(), parts.end(), 0.);
    return f.get();
}

int main() {
    int n = 1000;
    double x = .1;
    double result = run(n,10,x);
    std::cout << "Result is: " << result << std::endl;
    std::cout << "Difference of Taylor and C\texttt{++} result "
              << result - std::log1p(x) << " after "
              << n << " iterations." << std::endl;
}
```

all futures are ready. Thus, only the final call to `get()` can block.

For HPX, anything that would normally block will instead be suspended and switched out, similar to what C++ Coroutines would do.

Which leads us to the explicit coroutine code. Performing the suspend and resume operations are sure to introduce overheads, but they should not be as large as they seem to be from our data. This was easily the slowest version of the code. See Listing 3.

The parallel library approach does not attempt to suspend or resume, it performs a simple fork-join on evenly divided threads. This avoids the overheads of suspending and resuming, but potentially causes threads to wait unnecessarily at the joins. Listing 4 shows this approach.

Finally, senders and receivers, Listing 6 shows the most recent proposed method of implementing asynchrony in C++. This represents an effort to provide ways to express asynchrony while avoiding the overheads of futures and coroutines. Our data shows that it is fairly successful as, for most core counts, this was the fastest.

Let us apply this example to the C++ programming language. Figure 3 shows the classification of the C++ approaches concerning task-based and data parallelism. For parallelism, the C++ standard provides two approaches. First, the parallel algorithms introduced with the C++ 17 standard, see Section Parallel algorithms. And second,

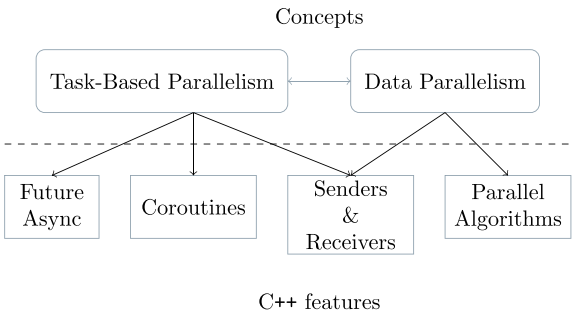

Fig. 3 On the top: The two concepts, namely, task-based and data parallelism. Where task-based parallelism enables arbitrary tasks to overlap, data parallelism enables parallel computation on arrays. On the bottom, the implementations for task-based and data parallelism in Modern C++: Futures + Async (Futures and Futurization); Parallel Algorithms (Section Parallel algorithms); Senders and Receivers (Section Senders and Receivers); and Coroutines + Async (Section Coroutines)

Senders and Receivers provides support for parallelism through its `bulk` function.

Note that the parallel algorithms are restricted since these algorithms can only operate on the elements of containers, e.g. `std::vector`. Some algorithms like `std::sort` or `std::find_if` are customizable by providing compare operators like `std::greater<double>()` or providing functions or lambda functions.

A more flexible option is to use C++'s support for concurrent programming, namely asynchronous programming

Listing 6 Implementation of the Taylor series of the natural logarithm using sender and receivers.

```
#include <hpx/execution.hpp>

using namespace hpx::execution::experimental;

template <typename T> concept sender = is_sender_v<T>;

namespace tt = hpx::this_thread::experimental;

---

double run(size_t n, size_t num_threads, double x) {
  thread_pool_scheduler sch{};

  size_t partition_size = n/num_threads;
  std::vector<double> partial_results(partition_size);

  sender auto s = schedule(sch) |
    bulk(num_threads, [&](auto i) {
      size_t begin = i * partition_size;
      size_t end = (i + 1) * partition_size;
      if (i == num_threads - 1) end = n;
      double partial_sum = 0;
      for(int i=begin; i <= end; i++) {
        double e = i+1;
        double term = std::pow(-1.0, e+1) * std::pow(x, e) / e;
        partial_sum += term;
      }
      partial_results[i] = partial_sum;
    }) |
    then([&]() {
      double sum = 0;
      for(int i=0;i<partition_size;i++)
        sum += partial_results[i];
      return sum;
    });
  auto [result] = *tt::sync_wait(std::move(s));
  return result;
}

int main() {
  double x = .1;
  double r = run(10000,10,x);
  double a = log(1+x);
  std::cout << "r=" << r << "   " « a« " => " « fabs(r-a) « std::endl;
```

using futures (and the majority of Senders and Receivers functionality). The interface `std::async` and `std::future` and their counterparts `hpx::async` and `hpx::future` provide higher-level alternatives to low-level programming using `std::threads` and `hpx::thread`, respectively.

Furthermore, HPX allows the programmer to combine parallel algorithms and asynchronous programming by asynchronously launching the algorithms that return a future. The API for `std::async` and `std::future` was introduced with the C++ 11 standard, but might be deprecated soon and be replaced with its successor senders and receivers. The current outline is to accept senders and receivers for the C++ 26 standard. However, HPX implements the latest proposal. See Section Senders and Receivers.

Coroutines were added with the C++ 20 standard and also support concurrent programming, see Section Coroutines. The `co_return`, `co_yield`, and `co_await` features were added to suspend and resume coroutines. Note that coroutines themselves do not provide parallelism per se and can be used to create a generator on a single core. Senders and receivers, curiously, provide features for task-based and data parallelism.

Figure 3 applies to HPX, since it implements all the above features of task-based and data parallelism in the standard. For a comparison of task-based and data parallelism in Chapel, Charm++, C++, HPX, Go, Julia, Python, Rust, Swift, and Java for a 1D heat equation solver, we refer to [31].

Table 2 Summary of CPU architectures, compilers, and dependencies used for the performance measurements in Fig. 4

CPU	CPUs	gcc	hpx	boost	hwloc	jemalloc/tcmalloc
Intel Xeon Gold 6140	2	9.2/12.2	1.8.1/1.9.1	1.78/ 1.82	2.2.0/ 2.9.1	5.2.0/ 2.1
AMD EPYC 7543	2	9.2/12.2	1.8.1/ 1.9.1	1.78/ 1.82	2.2.0/ 2.9.1	5.2.0/ 2.1
A64FX	1	12.1/ 12.2	1.8.1/1.9.1	1.78/ 1.82	2.2.0/2.9.1	5.2.0/ 2.1

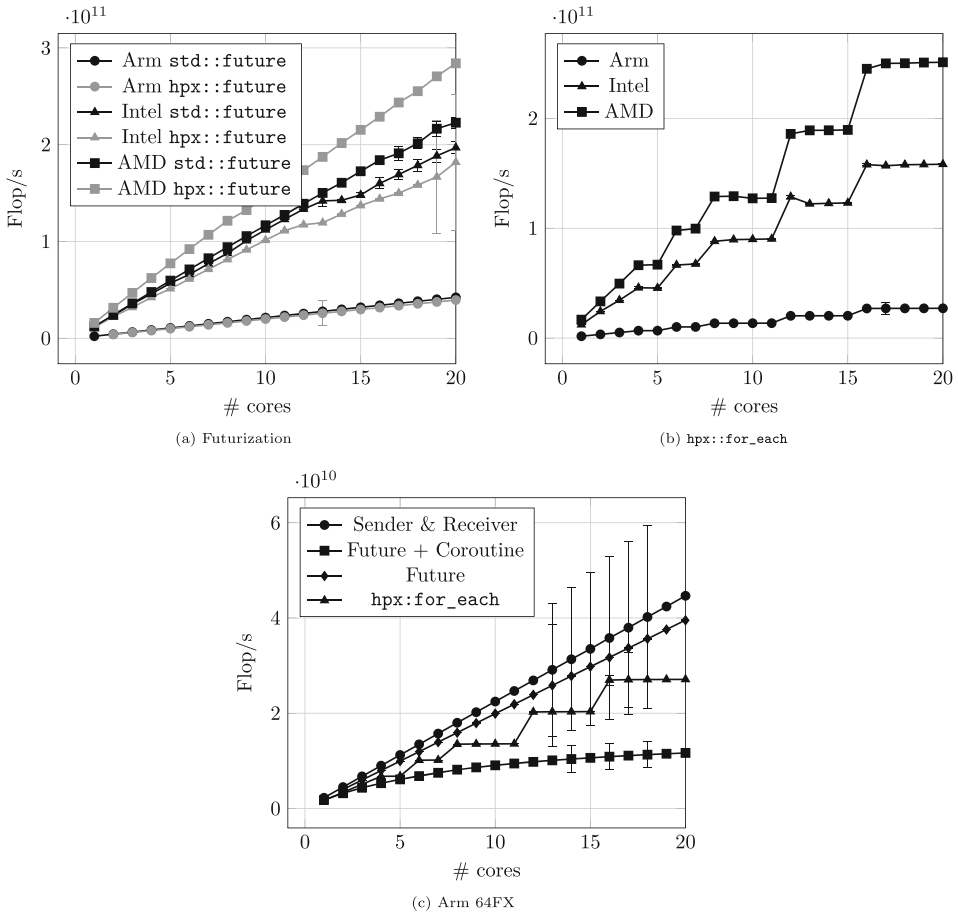

Fig. 4 The median out of ten runs with variations for various approaches for the Taylor series. Futurization using `std::future` and `hpx::future`(a) and HPX's parallel algorithm using `hpx::for_each`(b). On Arm64FX coroutines and sender & receiver were tested (c). To create an artificially work load, we computed the Taylor series in Equation (1) for $n = 1000000000$ and measured 100000028581 floating point operations using *perf* on a single Intel Core. Details on compilers and software versions are listed in Table 2

Performance Comparison

For performance measurements on different CPUs, we compiled all examples using gcc 12.1.0 for Arm, using gcc 9.2.0 for AMD and Intel. HPX 1.8.1 was compiled with the following dependencies: boost 1.78.0, hwloc 2.2.0, and jemalloc 5.2.0. We used different compilers on AMD and Intel, since these were the default compilers on the compute nodes. From our experience, the compiler version does not affect HPX's performance much. For the Mandelbrot set, we used the gcc 12.2.0 compiler on Arm A64FX. We installed the compiler gcc 12.2.0 on the AMD and Intel nodes using Spack [32]. On the Arm node the compiler was available as a Module file. We had to use HPX 1.9.1 to accommodate the support of a newer gcc versions. We used the following dependencies: boost 1.82.0, hwloc 2.9.1, and tcmalloc 2.1. Table 2 summarizes the versions of dependencies and CPU architectures used for the performance measurements in Fig. 4. For all core counts, the code was executed ten times and the median out of these runs is plotted. The error bars show the variances within these ten runs. For some approaches, we observe high variance for HPX on larger core counts.

Taylor Series

Figure 4 shows the performance obtained for all four of the programming mechanisms presented in this paper for the Taylor series: for ARM A64FX, AMD EPYC™ 7543, and Intel® Xeon® Gold 6140, respectively. To create an artificial work load, we computed the Taylor series in Equation (1) for $n = 1000000000$. We used *perf* on the Intel CPU to obtain the floating point operations of 100000028581 on a single

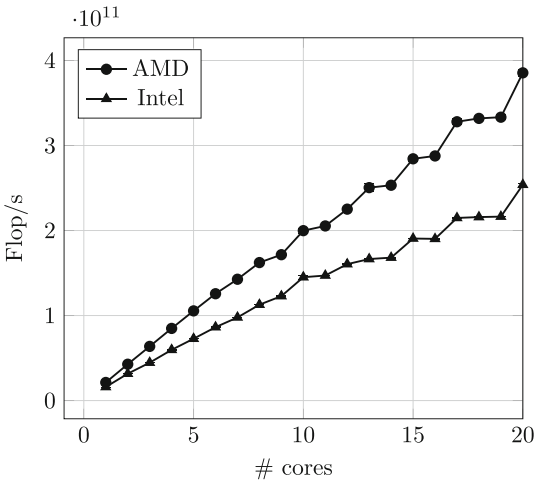

Fig. 5 Running the parallel algorithms in Fig. 4(b) using HPX's dynamic chunk size, see Section Additional HPX Features, to have a linear scaling and some slightly better performance. However, this feature is not yet in the C++ standard and is solely provided by HPX

core. For futures using `std::future` and `hpx::future` (a), we see that on Arm both implementations perform the same. Similar behavior is obtained for Intel. However, on AMD `hpx::future` performs better. Here, the overhead of using HPX is negligible. For more details on the overheads of HPX and Charm++, we refer to [9]. For HPX's parallel algorithms using `hpx::for_each`(b), AMD performed better than Intel and Arm is around one order of magnitude slower. The results on Arm64FX are shown in(c). The performance of the two more recent C++ features is one order of magnitude slower on Arm than on the two other architectures. Senders and receivers showed the best performance on Arm. However, one should not conclude that this paradigm is inherently faster based on this test. Note that we experience some high variation on higher node counts. More investigation is needed for this feature. For more performance measurements on Rikken's Supercomputer Fugaku, we refer to [33].

Additional HPX Features

For `hpx::for_each`, the performance in Fig. 4(b) on Intel and AMD is not a straight line. Instead, we observe some rolling hills. In this case, the default chunk size of one was used. Note that in the C++ standard there is currently no option to specify the chunk size yet. HPX, however, provides such an option, see Listing 5 in Section Additional HPX Features. Figure 5 shows the usage of the chunk sizes

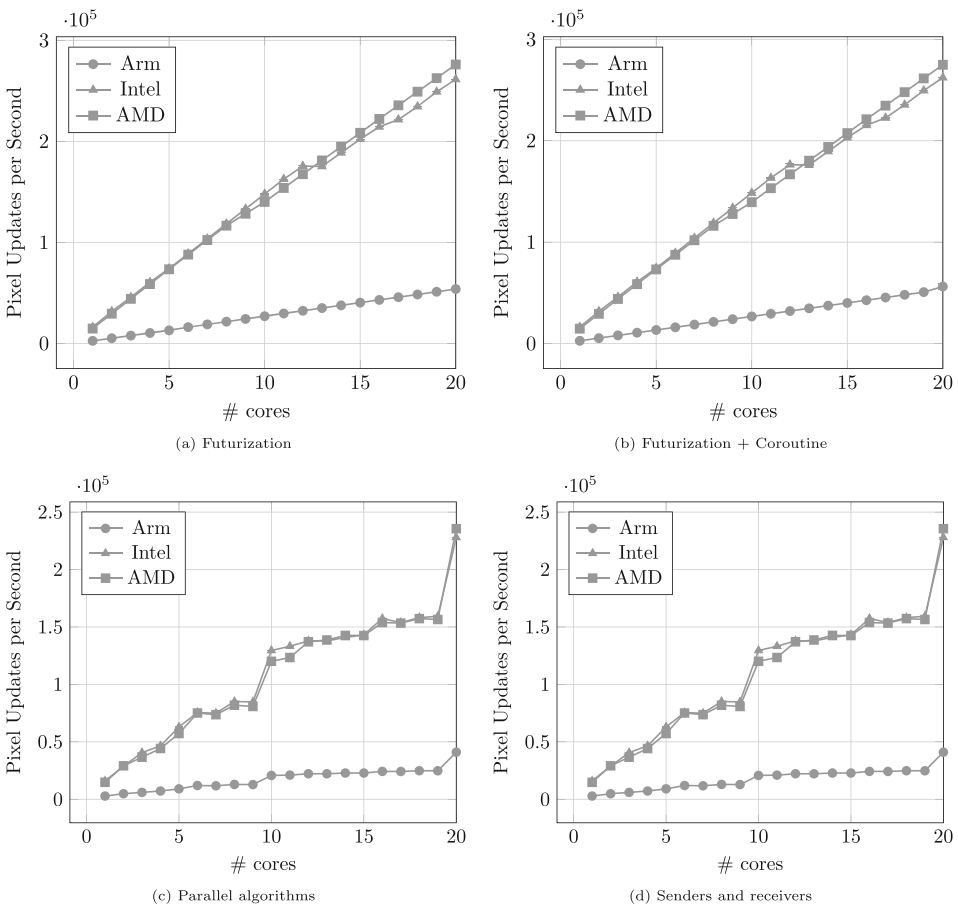

Fig. 6 The median out of ten runs with variations for methods of parallelizing the Mandelbrot set. Using `std::future` and `hpx::future` (a), futures + coroutines (b), HPX's parallel algorithm using `hpx::for_loop` (c), and senders and receivers (d). To create an artificial work load, we computed the Mandelbrot set in Equation (2) for 20000 × 20000 pixels

to make the scaling more linear by using a dynamic chunk `hpx::execution::dynamic_chunck_size` size of 1e6. Figure 5 shows the performance on Intel and AMD. For both architectures, the scaling behavior looks linear and the Flop\s are a little bit higher. The additional features provided by HPX can affect the performance. However, these features are not yet in the C++ standard.

Mandelbrot Set

Figure 6 shows the performance obtained for all four of the programming mechanisms presented in this paper for the Taylor series: for ARM A64FX, AMD EPYC™ 7543, and Intel® Xeon® Gold 6140, respectively. To create an artificial work load, we computed the Mandelbrot set in Equation (2) for 20000 × 20000 pixels. Figure 6a shows the pixel updated per seconds (PUPS) using `hpx::async` and `hpx::future`. Figure 6b shows the pixels updated per seconds (PUPS) using futurization + coroutines. Hare, AMD and Intel are comparable and ARM A64FX is slower. Figure 6c shows the pixels updated per seconds (PUPS) using HPX's parallel algorithms. Here, we see again the effect on the default chunk size of the parallel algorithm for Intel and AMD. Here, using a static or dynamic chunk size which is provided by HPX, but not yet in the C++ standard, could straighten the lines, see Section Additional HPX Features. The Intel and AMD results are close for this benchmark. Figure 6d shows the pixels updated per seconds (PUPS) using senders and receivers. Here, AMD and Intel are comparable and ARM A64FX is slower. For all approaches, we have seen that Arm A64FX had the lowest performance. AMD was comparable to Intel. This aligns with the results for the Taylor series in Section Taylor Series. However, for this benchmark with more work in the computations, we observed less variance in the measurements.

Conclusion

We have shown that Modern C++, through its standard libraries and language features, provides a complete and expressive shared memory parallel programming infrastructure for a single node. Therefore, no external libraries or language extensions are necessary to write high-quality parallel C++ applications. We sketched an example of how to use futures, coroutines, and parallel algorithms in the current C++ standard based on a Taylor series code. Furthermore, we provided an introduction to senders and receivers, a framework that might be available in a future C++ standard. For most of these programming mechanisms, we showcased the implementation using the C++ Standard Library using system threads and using the C++ Library for Concurrency and Parallelism (HPX). We did this because HPX provides a cutting-edge implementation of the parallel library proposals being considered by the C++ standards committee.

A performance comparison on an Intel® CPU, AMD CPU, and ARM® A64FX demonstrates that the proposed parallel programming mechanisms do achieve portability without code changes for the Taylor series and the Mandelbrot set.

Acknowledgements We would also like to thank Alireza Kheirkhahan and the HPC admins who support the Deep Bayou cluster at Louisiana State University.

Funding The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a $5 M National Science Foundation grant (#1927880).

Data availability The code for all examples is available on GitHub®[2] or Zenodo™,[3] respectively.

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

Research involving human participants and/or animals This article does not contain any studies with human participants performed by any of the authors.

Informed consent Not applicable, since no humans were involved in our research.

Additional Code Listing

The implementation of the Mandelbrot set is shown for futures in Listing 7, coroutines in Listing 8, parallel algorithms in Listing 9, and senders and receivers in Listing 10, respectively.

[2] https://github.com/STEllAR-GROUP/parallelnumericalintegration.

[3] https://zenodo.org/record/7515618.

Listing 7 Parallel implementation of the Mandelbrot set using `hpx::async` and `hpx::future`.

```
typedef std::complex<double> complex;
```

```
void launch(size_t i, size_t pixel_x, size_t pixel_y, PBM* pbm) {
  complex c =
      complex(0, 4) * complex(i, 0) / complex(pixel_x, 0) - complex(0, 2);

  for (size_t j = 0; j < pixel_y; j++) {
    int value = compute_pixel(c + 4.0 * j / pixel_y - 2.0);
    std::tuple<size_t, size_t, size_t> color = get_rgb(value);
    pbm->row(i)[j] =
        make_color(std::get<0>(color), std::get<1>(color), std::get<2>(color
            ));
  }
}
```

```
int pixel_x = std::stoi(argv[1]);
int pixel_y = std::stoi(argv[2]);

PBM pbm(pixel_x, pixel_y);

auto start = std::chrono::high_resolution_clock::now();

std::vector<hpx::future<void>> futures;

for (size_t i = 0; i < pixel_x; i++) {
    futures.push_back(std::move(hpx::async(launch, i, pixel_x, pixel_y, &pbm
        )));
  }

hpx::when_all(futures).get();

auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> diff = end - start;
std::cout << pixel_x * pixel_y << "," << diff.count() << std::endl;

pbm.save("image_future.pbm");
```

Listing 8 Parallel implementation of the Mandelbrot set using futures + coroutines.

```
#include <coroutine>

typedef std::complex<double> complex;

---

hpx::future<void> run(size_t pixel_x, size_t pixel_y, PBM* pbm) {
  std::vector<hpx::future<void>> futures;

  for (size_t i = 0; i < pixel_x; i++) {
    futures.push_back(std::move(hpx::async([i, pixel_x, pixel_y, &pbm]() {
      complex c =
          complex(0, 4) * complex(i, 0) / complex(pixel_x, 0) - complex(0, 2);

      for (size_t j = 0; j < pixel_y; j++) {
        int value = compute_pixel(c + 4.0 * j / pixel_y - 2.0);
        std::tuple<size_t, size_t, size_t> color = get_rgb(value);
        pbm->row(i)[j] = make_color(std::get<0>(color), std::get<1>(color),
                                    std::get<2>(color));
      }
    })));
  }

  auto futures2 = co_await hpx::when_all(futures);

  co_return;
}

---

int pixel_x = std::stoi(argv[1]);
int pixel_y = std::stoi(argv[2]);

PBM pbm(pixel_x, pixel_y);

auto start = std::chrono::high_resolution_clock::now();

run(pixel_x, pixel_y, &pbm).get();

auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> diff = end - start;
std::cout << pixel_x * pixel_y << "," << diff.count() << std::endl;

pbm.save("image_future_coroutine.pbm");
```

Listing 9 Parallel implementation of the Mandelbrot set using HPX's parallel algorithms.

```
#include <hpx/parallel/algorithm.hpp>

typedef std::complex<double> complex;
---

int pixel_x = std::stoi(argv[1]);
int pixel_y = std::stoi(argv[2]);

PBM pbm(pixel_x, pixel_y);

auto start = std::chrono::high_resolution_clock::now();

hpx::experimental::for_loop(
      hpx::execution::par, 0, pixel_x, [pixel_x, pixel_y, &pbm](size_t i) {
        complex c =
            complex(0, 4) * complex(i, 0) / complex(pixel_x, 0) - complex(0,
                2);

        for (size_t j = 0; j < pixel_y; j++) {
          int value = compute_pixel(c + 4.0 * j / pixel_y - 2.0);
          std::tuple<size_t, size_t, size_t> color = get_rgb(value);
          pbm.row(i)[j] = make_color(std::get<0>(color), std::get<1>(color),
                                     std::get<2>(color));
        }
      });

auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> diff = end - start;
std::cout << pixel_x * pixel_y << "," << diff.count() << std::endl;

pbm.save("image_par.pbm");
```

Listing 10 Parallel implementation of the Mandelbrot set using senders and receivers.

```
#include <hpx/execution/algorithms/sync_wait.hpp>
#include <hpx/execution_base/sender.hpp>

using namespace hpx::execution::experimental;
typedef std::complex<double> complex;

template <typename T>
concept sender = is_sender_v<T>;

namespace tt = hpx::this_thread::experimental;
```

```
void run(size_t pixel_x, size_t pixel_y, PBM* pbm) {
  thread_pool_scheduler sch{};

  sender auto s =
      schedule(sch) | bulk(pixel_x, [pixel_x, pixel_y, &pbm](auto i) {
        complex c =
            complex(0, 4) * complex(i, 0) / complex(pixel-x, 0) - complex(0,
                2);

        for (size_t j = 0; j < pixel_y; j++) {
          int value = compute_pixel(c + 4.0 * j / pixel_y - 2.0);
          std::tuple<size_t, size_t, size_t> color = get_rgb(value);
          pbm->row(i)[j] = make_color(std::get<0>(color), std::get<1>(color)
              ,
                                      std::get<2>(color));
        }
      });

  *tt::sync_wait(std::move(s));
}
```

```
int pixel_x = std::stoi(argv[1]);
int pixel_y = std::stoi(argv[2]);

PBM pbm(pixel_x, pixel_y);

auto start = std::chrono::high_resolution_clock::now();

run(pixel_x, pixel_y, &pbm);

auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> diff = end - start;
std::cout << pixel_x * pixel_y << "," << diff.count() << std::endl;

pbm.save("image_sender_receiver.pbm");
```

References

1. Butenhof D.R. Programming with POSIX threads (Addison-Wesley Professional, 1997)
2. C++ Standards Committee, ISO/IEC 14882:2011, Standard for Programming Language C++ (C++11). Tech. rep., ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee) (2011). https://wg21.link/N3337, last publicly available draft
3. C++ Standards Committee, ISO/IEC 14882:2014, Standard for Programming Language C++ (C++14). Tech. rep., ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee) (2011). https://wg21.link/N4296, last publicly available draft
4. The C++ Standards Committee, ISO International Standard ISO/IEC 14882:2017, Programming Language C++. Tech. rep., Geneva, Switzerland: International Organization for Standardization (ISO). (2017). http://www.open-std.org/jtc1/sc22/wg21
5. Chandra R, Dagum L, Kohr D, Menon R, Maydan D, McDonald J. Parallel programming in OpenMP (Morgan kaufmann, 2001)
6. Leiserson C.E. Cilk (Springer US, Boston, MA, 2011), pp. 273–288. https://doi.org/10.1007/978-0-387-09766-4_289.
7. Edwards HC, et al. Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J Parallel Distrib Comput. 2014;74(12):3202–16.
8. Kale LV, Krishnan S. In Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, 1993;pp. 91–108.
9. Wu N, Gonidelis I, Liu S, Fink Z, Gupta N, Mohammadiporshokooh K, Diehl P, Kaiser H, Kale L.V. in Euro-Par 2022: Parallel Processing Workshops: Euro-Par 2022 International Workshops, Glasgow, UK, August 22–26, 2022, Revised Selected Papers (Springer, 2023), 5–16
10. Chamberlain B.L. et al., Parallel programmability and the chapel language. The International Journal of High Performance Computing Applications 2007;21(3)
11. Ebcioglu K. et al., in Proceedings of the International Workshop on Language Runtimes, OOPSLA, 30 (Citeseer, 2004)
12. Zheng Y. et al., in 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IEEE, 2014), 1105–1114
13. Thoman P. et al., A taxonomy of task-based parallel programming technologies for high-performance computing. The Journal of Supercomputing 2018;74(4)
14. Kaiser H, et al. HPX - The C++ Standard Library for Parallelism and Concurrency. Journal of Open Source Software. 2020;5(53):2352.
15. Daiß G, Amini P, Biddiscombe J, Diehl P, Frank J, Huck K, Kaiser H, Marcello D, Pfander D, Pfüger D. in Proceedings of the international conference for high performance computing, networking, storage and analysis 2019: 1–37
16. Yan J, Kaiser H, Snir M. in Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (Association for Computing Machinery, New York, NY, USA, 2023), SC-W '23, 1151–11https://doi.org/10.1145/3624062.3624598.
17. Bonachea D, Hargrove P.H. in International Workshop on Languages and Compilers for Parallel Computing (Springer, 2018), 138–158
18. Diehl P, Brandt S.R, Kaiser H. Parallel C++ – Efficient and Scalable High-Performance Parallel Programming Using HPX, vol. 1 (Springer Cham, 2024), 240
19. Marcello DC, Shiber S, De Marco O, Frank J, Clayton GC, Motl PM, Diehl P, Kaiser H. Octo-Tiger: a new, 3D hydrodynamic code for stellar mergers that uses HPX parallelisation. Monthly Notices of the Royal Astronomical Society. 2021. https://doi.org/10.1093/mnras/stab937.
20. Diehl P, Daiss G, Brandt S, Kheirkhahan A, Kaiser H, Taylor C, Leidel J. in Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (Association for Computing Machinery, New York, NY, USA, 2023), SC-W '23, 1533–154https://doi.org/10.1145/3624062.3624230.
21. Diehl P, Dais G, Huck K, Marcello D, Shiber S, Kaiser H, Pfluger D. in 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (IEEE Computer Society, Los Alamitos, CA, USA, 2023), 682–69https://doi.org/10.1109/IPDPSW59300.2023.00116. https://doi.ieeecomputersociety.org/10.1109/IPDPSW59300.2023.00116
22. Diehl P, Daiß G, Marcello D, Huck K, Shiber S, Kaiser H, Frank J, Clayton G.C, Pflüger D. in 2021 IEEE International Conference on Cluster Computing (CLUSTER) (IEEE, 2021), 204–214
23. Mandelbrot BB. Fractal aspects of the iteration of $z \rightarrow \lambda z$ (1-z) for complex λ and z. Ann N Y Acad Sci. 1980;357(1):249–59.
24. Brooks R, Matelski J.P. in Riemann surfaces and related topics: Proceedings of the 1978 Stony Brook Conference, 1 (Princeton University Press Princeton, New Jersey, 1981)
25. Dominiak M. et al. std::execution (2022). https://wg21.link/p2300
26. Grubel P. et al., in 2015 IEEE International Conference on Cluster Computing (IEEE, 2015),682–689
27. Shirzad S. et al., in 2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) (IEEE, 2019) 31–43
28. Khatami Z. et al., in Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware 2017:1–8
29. Yadav S. et al., in 2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2) 2021: 20–29
30. Kretz M, Lindenstruth V. Vc: A C++ library for explicit vectorization. Software: Practice and Experience. 2012;42(11):1409–30.
31. Diehl P, Brandt S.R, Morris M, Gupta N, Kaiser H. Benchmarking the parallel 1d heat equation solver in chapel, charm++, c++, hpx, go, julia, python, rust, swift, and java 2023.
32. Gamblin T, LeGendre M, Collette M.R, Lee G.L, Moody A, de Supinski B.R, Futral S. in SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis (IEEE Computer Society, Los Alamitos, CA, USA, 2015), 1–1https://doi.org/10.1145/2807591.2807623. https://doi.ieeecomputersociety.org/10.1145/2807591.2807623
33. Diehl P, Daiß G, Huck K.A, Marcello D, Shiber S, Kaiser H, Pflüger D in IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023 - Workshops, St. Petersburg, FL, USA, May 15-19, 2023 (IEEE, 2023), 682–69https://doi.org/10.1109/IPDPSW59300.2023.00116.