ADVANCED PARALLEL PROGRAMMING IN C++

A PREPRINT

Patrick Diehl, Steven R. Brandt, and Hartmut Kaiser
Center of Computation & Technology
Louisiana State University

November 30, 2021

ABSTRACT

Many applied mathematics codes are based on the C++ programming language. However, many of these codes do not utilize modern C++ features, especially the features from the recent C++ 17 standard for parallel computations. These features can significantly simplify parallel computations using multiple cores. This review briefly introduces asynchronous programming facilities introduced in the C++ 11 standard and the parallel algorithms introduced in the C++ 17 standard. With these two paradigms, C++ code can be accelerated using multiple cores using the C++ standard without any need for some external special purpose libraries, like *e.g.* OpenMP. The C++20 standard made additional and valuable contributions to the C++ api for parallelism and concurrency. While these may still be unavailable in some compilers, they are already implemented in HPX [4].

1 Asynchronous programming

The C++11 standard [7] has introduced asynchronous task programming. Here, the concept of futurization using `std::future` was introduced. A future is a placeholder for the result of an asynchronous function (launched e.g. using `std::async()`). The runtime promises that once the computation of the task passed to `std::async()` has finished, the future will receive the result. One can use the function `.get()` to access the result. If the computation has finished the result will be returned. If the computation is still ongoing, the function will suspend the current thread until the future has become ready, at which point the result is returned to the user. Let us look at the following Taylor expansion for the natural logarithm

$$\ln(x) = \sum_{n=1}^{N} \frac{(-1)^{n+1}}{n}(x-1)^n. \tag{1}$$

Listing 1 shows the asynchronous computation of the Taylor series. First, a function `taylor_part` that computes a part of the sum is defined in Line 5. In Line 20 the sum of length n is separated into two partitions of $^n/_2$ and each of them is asynchronously launched, and a future is returned. Since these two function launches run concurrently on two cores, we need to synchronize their execution and collect the results. In Line 25 the `.get()` function is called to access the result of each of the futures. Note that the code will introduce a barrier and waits until the first and second future are ready.

The C++ standard library for parallelism and concurrency (HPX) [4] implements the same functionality as mandated by the C++ standard. However, additional, more advanced synchronization facilities are provided by HPX. The function `.then()` provides a function similar to `.get()`, however it takes a function that is only to be run after the future's value arrives. While slightly less intuitive to program, this method avoids blocking. Also, it is possible to use `when_all()`, which waits for one or more futures. Once all futures are ready, `when_all()` returns a future that becomes ready only after all argument futures have become ready. Listing 2 uses these features.

Since the C++20 standard for coroutines has emerged, HPX futures have become awaitable. That means that the less natural style of programming using `.then()` can be replaced by calls to `co_await`. Listing 3 illustrates how to use this style of programming with HPX. The call to `co_await` suspends execution of the current thread until the value in the future is ready. Therefore, `co_await` can return the value directly.

Listing 1: Asynchronous computation of the Taylor expansion for the natural logarithm.

```
#include <cmath>
#include <future>
#include <iostream>

double taylor_part(double x, size_t start, size_t end){

    double result = 0;
    for(size_t i = start; i < end; i++)
    {
        result += std::pow(-1,i+1)/i * std::pow(x-1,i);
    }
    return result;
}

int main(void){

    double x = 0.5;
    size_t n = 100;

    // Launch the function async
    std::future<double> f1 = std::async(taylor_part,x,1,n/2);
    std::future<double> f2 = std::async(taylor_part,x,n/2,n);

    // Gather the result
    std::cout << f1.get() + f2.get() << std::endl;
    return EXIT_SUCCESS;
}
```

2 Parallel algorithms

A common opportunity for shared-memory parallel programming in C++ is Open Multi-Processing (OpenMP) [1]. Let us look at the implementation of computing the square root of all elements in a vector v of size n

$$v_i = \sqrt{v_i},\ \forall i \in 1, \dots, n \tag{2}$$

in a parallel fashion using OpenMP (see Listing 4). In Line 12, a `std::vector` is generated with length $n = 10000$ and in Line 15 the vector is filled with random numbers using the algorithms provided by the C++ standard library. In Line 18 a `for` loop is used to iterate over the elements of the vector. Now, the `#pragma`-based OpenMP API is used to parallelize the `for` loop. In Line 17 the `#pragma` to execute the loop in parallel is added. However, an additional API is needed to achieve the shared-memory parallelism.

In the C++ 17 standard [8], parallel algorithms were introduced. The shared-memory parallelism they provide is based on the Thread Building Blocks library (TBB) [6]. While it is possible to write parallel code directly using the C++ standard, and there is no need to use a second API, the C++ standard library contains 69 algorithms which work on standard containers, e.g. `std::vector` or `std::list`. Some versions of C++ libraries may not implement all of them. Also in C++ 17, execution policies were introduced to these 69 algorithms, allowing them to execute in parallel. Depending on the version of your C++ libraries, this feature may also be incomplete.

In this next listing, we will revise the previous example to use parallel algorithms. See Listing 5. Up to Line 14, the code is unchanged, except for the new header file `#include <execution>`, which is needed for the execution policies. In Line 16, a vector containing the indices of each vector element is generated. In Line 20 the OpenMP parallel `for` loop is replaced by `std::for_each` from the C++ standard library algorithms. Note that this was possible before. However, the first argument of this function, the execution policy, defines that this algorithm is to be executed in parallel. The C++ 17 standard introduced the following execution policies:

- **Sequential execution**:
  By adding `std::execution::seq` the algorithm is executed sequentially using one thread as in the previous C++ standards.
- **Parallel execution**:
  By adding `std::execution::par` the algorithm is executed in parallel using all available threads.

Listing 2: Asynchronous computation of the Taylor expansion for the natural logarithm using HPX.

```
#include <hpx/hpx_main.hpp>
#include <hpx/future.hpp>
#include <cmath>
#include <iostream>

double taylor_part(double x, size_t start, size_t end){

    double result = 0;
    for(size_t i = start; i < end; i++)
    {
        result += std::pow(-1,i+1)/i * std::pow(x-1,i);
    }
    return result;
}

int main(void){

    double x = 0.5;
    size_t n = 100;
    size_t partitions = 3;
    std::vector<hpx::future<double>> futures;
    futures.reserve(partitions);

    // Launch the function async
    futures.push_back(hpx::async(taylor_part,x,1,n/3));
    for(size_t i = 1; i < partitions; i++)
      futures.push_back(hpx::async(taylor_part,x,i*n/3,(i+1)*n/3));

    // Gather the result
    hpx::when_all(futures).then([](auto&& f){
        std::vector<hpx::future<double>> futures = f.get();
        double result = 0;
        for(size_t i = 0; i < futures.size(); i++)
            result += futures[i].get();
        std::cout << result << std::endl;
        });
    return EXIT_SUCCESS;
}
```

- **Vectorized parallel execution**:
  By adding `std::execution::par_unseq` the algorithm is executed in parallel but in addition vectorization is used.

By specifying the execution policy in Line 21, the algorithm is easily parallelized. Note that this was implemented using only the C++ standard and not an external tool like OpenMP.

Next, we will look at an example that computes the sum s of all elements in a vector v with n elements

$$s = \sum_{i=0}^{n} v_i, \tag{3}$$

see Listing 6. In Line 6 a `std::vector` with length `n` is generated. The first algorithm of the C++ Standard Library `std::fill` is used to fill all elements with negative one, see Line 11. Finally, the second algorithm `std::accumulate` is used to compute the sum of all elements. Note that the naive approach would be to use a `for` loop to iterate over all the elements. In Line 17 the result is printed for validation. Again, one can use the execution policies to easily parrallelize this code by just adding the execution policy in Line 14 and Line 17, respectively. The corresponding code is shown in Listing 7.

These two small examples show only a few of the parallel algorithms of the C++ 17 standard, which are experimentally supported by the GNU compiler collection (gcc) ≥ 9 and the Microsoft Visual Studio compiler (MVSC). However, the C++ standard library for parallelism and concurrency (HPX) implements all the defined features in the C++ 17 standard. Note that you can replace the `std::` name space by `hpx::` name space, since HPX strictly enhances the

Listing 3: Asynchronous computation of the Taylor expansion for the natural logarithm using HPX.

```
#include <hpx/hpx_main.hpp>
#include <hpx/future.hpp>
#include <cmath>
#include <iostream>

double taylor_part(double x, size_t start, size_t end){

    double result = 0;
    for(size_t i = start; i < end; i++)
    {
        result += std::pow(-1,i+1)/i * std::pow(x-1,i);
    }
    return result;
}

// Can only use co_await inside a function that returns a future
hpx::future<double> get_taylor_part(double x, size_t n, size_t partitions) {
    double result = 0;
    std::vector<hpx::future<double>> futures;
    futures.reserve(partitions);

    // Launch the function async
    futures.push_back(hpx::async(taylor_part,x,1,n/3));
    for(size_t i = 1; i < partitions; i++)
      futures.push_back(hpx::async(taylor_part,x,i*n/3,(i+1)*n/3));

    // Gather the results
    auto futures2 = co_await hpx::when_all(futures);

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

    co_return result;
}

int main(void){
    double x = 0.5;
    size_t n = 100;
    size_t partitions = 3;
    std::cout << get_taylor_part(x,n,partitions).get() << std::endl;
    return EXIT_SUCCESS;
}
```

C++ 17 standard. We will show two small examples with new features which are not yet implemented in the major implementations. First, HPX provides some easier way to iterate over a vector without generating a vector of indices using `std::for_each`, see Listing 8. Listing 8 shows the simplification for iterating over the vector. In the previous example, we had to generate the vector of indices, which increases the memory consumption of the program. In Line 18, we use a `hpx::for_loop` which is more like the `for` loop where the start index is provided as the second argument and the end index as the third argument. In the lambda function, we can access the index, i which will go from zero up to `v.size()`. In addition, the parallel algorithms can be combined with the concept of futurization to obtain an `hpx::future`. Note that this is not provided by the C++ 17 standard.

3 Conclusions

In this article we have shown that it is possible to write parallel C++ programs that take advantage of features in the C++17 and C++20 standards, both to provide task parallelism and loop parallelism (with the parallel algorithms). In addition, we've shown how to use more advanced parallel features only available in the HPX programming framework.

Listing 4: Example to compute the element-wise square root of a vector using OpenMP.

```cpp
#include <cmath>
#include <vector>
#include <numeric>
#include <random>
#include <algorithm>

int main(void){

    size_t n = 10000;

    // Generate the vector with length n
    std::vector<double> v(n);

    // Initialize the vector with -1
    std::generate(v.begin(),v.end(),std::rand);

    #pragma omp parallel for
    for (size_t i = 0 ; i < v.size(); i++)
        v[i] = std::sqrt(i);

    return EXIT_SUCCESS;
}
```

Listing 5: Example: Compute the element-wise square root of a vector using C++.

```cpp
#include <cmath>
#include <vector>
#include <numeric>
#include <random>
#include <algorithm>
#include <execution>

int main(void){

    size_t n = 10000;
    // Generate the vector with length n
    std::vector<double> v(n);
    // Initialize the vector with -1
    std::generate(v.begin(),v.end(),std::rand);

    // Fill a new vector containing the indices
    std::vector<size_t> i = std::vector<size_t>(n);
    std::iota(std::begin(i), std::end(i), 0);

    std::for_each(
        std::execution::par,
        i.begin(),
        i.end(),
        [&](auto&& item)
        {
            v[item] = std::sqrt(v[item]);
        });

    return EXIT_SUCCESS;
}
```

Listing 6: Example to compute the sum of all elements in the vector using the C++ standard library.

```
#include <iostream>
#include <vector>
#include <numeric>

int main(void){

    size_t n = 10000;
    // Generate the vector with length n
    std::vector<int> v(n);
    // Initialize the vector with -1
    std::fill(n2.begin(),n2.end(),-1);

    // Compute the sum of all elements
    int s = std::accumulate(v.begin(),v.end(),0.0);

    // Output the result
    std::cout << "Result=␣" << result << std::endl;

    return EXIT_SUCCESS;
}
```

Listing 7: Example to compute the sum of all elements in the vector using the parallel algorithms provided by the C++ 17 standard.

```
#include <iostream>
#include <vector>
#include <numeric>
#include <execution>

int main(void){

  constexpr size_t n = 10000;

  // Generate the vector with length n
  std::vector<int> v(n);

  // Initialize the vector with -1
  std::fill(std::execution::par,v.begin(),v.end(),-1);

  // Compute the sum of all elements
  int s = std::reduce(std::execution::par,v.begin(),v.end(),0.0);

  // Output the result
  std::cout << "Result=␣" << result << std::endl;

  return EXIT_SUCCESS;
}
```

Moreover, we have identified two science applications which base their parallelism on this functionality. First, PeriHPX [3, 2] a non-local continuums mechanics code using all these features for its shared memory implementation. Second, Octo-Tiger [5] a multi-physics astrophysics code for simulating stellar mergers.

Supplementary materials

The source code of all the examples is available on GitHub[1] or Zenodo [], respectively. The build system CMake is used to compile all the examples.

[1]https://github.com/diehlpk/modern-cpp-examples

Listing 8: Example to compute the sum of all elements in the vector using the parallel algorithms provided by HPX.

```
#include <hpx/hpx_main.hpp>
#include <hpx/algorithm.hpp>
#include <hpx/future.hpp>

#include <iostream>
#include <vector>
#include <numeric>

int main(void) {
  constexpr size_t n = 10000;

  // Generate the vector with length n
  std::vector<int> v(n);

  // Initialize the vector with -1
  hpx::ranges::fill(hpx::execution::par, v, -1);

  // Compute the sum of all elements
  hpx::future<double> f = hpx::ranges::reduce(
      hpx::execution::par(hpx::execution::task), v, 0.0);

  // Output the result
  std::cout << "Result=␣" << f.get() << std::endl;

  return EXIT_SUCCESS;
}
```

Acknowledgments

I would like to acknowledge the students of the course "Parallel Computational Math" offered at the Mathematics Department at Louisiana State University for their feedback while teaching these modern C++ features as part of the course.

References

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

[2] Patrick Diehl, Prashant K Jha, Hartmut Kaiser, Robert Lipton, and Martin Lévesque. An asynchronous and task-based implementation of peridynamics utilizing hpx—the c++ standard library for parallelism and concurrency. *SN Applied Sciences*, 2(12):1–21, 2020.

[3] Prashant K Jha and Patrick Diehl. Nlmech: Implementation of finite difference/meshfree discretization of nonlocal fracture models. *Journal of Open Source Software*, 6(65):3020, 2021.

[4] Hartmut Kaiser, Patrick Diehl, Adrian S Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R Brandt, Nikunj Gupta, Thomas Heller, et al. Hpx-the c++ standard library for parallelism and concurrency. *Journal of Open Source Software*, 5(53):2352, 2020.

[5] Dominic C Marcello, Sagiv Shiber, Orsola De Marco, Juhan Frank, Geoffrey C Clayton, Patrick M Motl, Patrick Diehl, and Hartmut Kaiser. octo-tiger: a new, 3D hydrodynamic code for stellar mergers that uses hpx parallelization. *Monthly Notices of the Royal Astronomical Society*, 504(4):5345–5382, 04 2021.

[6] Chuck Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298, April 2008.

[7] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2011, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2011. http://www.open-std.org/jtc1/sc22/wg21.

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