Adapting the Parallel Algorithms in HPX for Usage with Senders and Receivers

Proposal for the Google Summer of Code 2024

PROJECT PROPOSAL

C++ senders and receivers (S/R), which were introduced in proposal P0443 (*A Unified Executors Proposal for C++*) and refined in subsequent proposals like P2300, are intended to provide a uniform model for asynchronous programming in Standard C++. It aims at providing a composable and generic, yet importantly, foolproof approach to writing parallel and asynchronous programs, all while ensuring developers retain control over the precise execution parameters.

As of now, there are several experimental implementations of the facilities proposed in P2300, most notably Meta's libunifex and NVIDIA's reference implementation. HPX also offers its own implementation of senders and receivers. Importantly for this project, HPX provides internal capabilities to adapt existing parallel algorithm implementations to work with senders and receivers, in particular the `tag_parallel_algorithm` class, which conveniently wraps existing algorithm CPOs to enable their use as sender adapters.

However, not all parallel algorithms in HPX currently support the S/R interface. For instance, the `ends_with` or the two `shift_*` algorithms lack implementations based on `tag_parallel_algorithm`. Additionally, some algorithms have an interface compatible with S/R (through `tag_parallel_algorithm`), but lack S/R support in their underlying implementation, since they rely on one of two partitioners, which have not yet been adapted to S/R (`partitioner_with_cleanup` and `scan_partitioner`). Furthermore, it seems that there are still some bugs that need to be addressed, since attempting to use the S/R versions of some algorithms will lead to compilation errors, for example the `adjacent_find` or the `copy` algorithms. (A minimal example for this latter can be found here.)

Completing the implementation of the S/R support of the parallel algorithms – the principal goal of this project – would benefit HPX by advancing the experimental implementation of S/R in HPX, which would align well with HPX's ambitions of staying current with important C++ standardization proposals and providing a testing ground for novel concepts introduced there.

PROJECT PLAN

To execute the project, I would propose the following steps and timeline:

MILESTONE 1: ADDING TEST CASES AND BUG FIXING *May 27 – Jun 23*

The initial step would involve adding unit test cases for the S/R versions of the algorithms, where they are missing yet. This will help identify any potential issues similar to those described earlier and may serve as an aid for bug fixing. The primary objective of this milestone is to ensure all existing adapted algorithms function reliably with S/R. It is the first step, because any insights gained during it might be relevant when adapting the remaining algorithms in the subsequent steps.

MILESTONE 2: ADAPTING THE PARTITIONERS *Jun 24 – Jul 14*

While the regular `partitioner` and `foreach_partitioner` were adapted to work with S/R, this is not the case for the `partitioner_with_cleanup` and `scan_partitioner`. Therefore, in order to be able to use the algorithms, which rely on one of those two partitioners, with S/R it is required to adapt them in the same way, as it was done in the regular and `foreach_partitioner`.

MILESTONE 3: ADAPTING THE REMAINING ALGORITHMS *Jul 15 – Aug 4*

This step would entail adding a `tag_parallel_algorithm`-based CPO for every algorithm, that does not have such a wrapper yet, so that these remaining parallel algorithms can also be used as sender adapters. Additionally, this would also encompass adding unit test cases for the newly implemented S/R versions of the algorithms.

MILESTONE 4: ADDING PERFORMANCE TESTS *Aug 5 – Aug 19*

As a final step, performance tests would be added for the S/R versions of selected algorithms, in order to enable an efficiency comparison with their regular counterparts. This could be achieved by leveraging existing benchmarks and substituting algorithm calls in those with equivalent S/R expressions. Additionally, time permitting, an optional goal could be extending performance tests to algorithms lacking preexisting benchmarks, which would also require the addition of performance tests for other versions of the examined algorithms.

This timeline is visualized in the chart in figure 1.

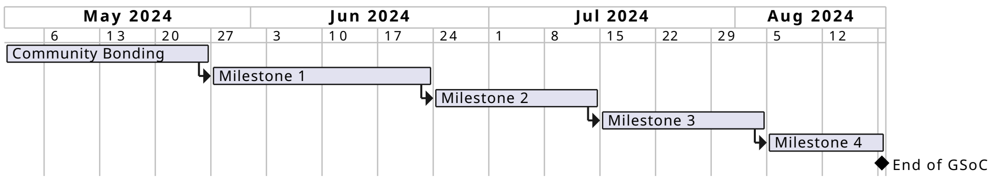

Figure 1: Proposed Project Timeline

PERSONAL INFORMATION

Regarding my availability, I am committed to dedicate a large part of this year's summer to the project, so that I can contribute at least 350 hours, equivalent to the scope of a large project. I aim to adhere to the standard timeline of the Google Summer of Code, so starting around May 1st and concluding around August 26th. However, my university courses run until the end of June, with final examinations expected to take place in the first week of July. Therefore, I suspect that the time, which I can spend on the project, might be betimes limited in May and especially June, which is why I aim to focus on the project in July and August, working on the project 'full-time' during those two months.

Background

As a third-year undergraduate student majoring in Computer Science, I have taken courses in discrete mathematics and calculus, programming and software engineering classes, which were largely focussed on Java, but also included a fair amount of Python and C++, as well as classes about algorithms and data structures. During my elective coursework, I could deepen my understanding of algorithm theory, including proof techniques and basic algorithm design principles. Additionally, I could learn more about parallel and distributed computing, gaining some familiarity with tools such as OpenMP and MPI, along with the parallelism and concurrency support offered by the C++ Standard Template Library (e.g. std::thread, std::future, std::atomic, std::lock, etc.), and how to apply these concepts to simple practical problems, such as parallelizing the approximation of the 2-dimensional heat equation using the five-point stencil.

Much of my programming experience stems from university courses, where programming often played a significant role. This does not only hold for the ones focused solely on software engineering, but also for those concerned with other topics such as mathematics, optimization, and statistics. Moreover, I could gain practical experience through occasional contract work as a programmer for a local recycling company, where I supported the automatization and simplification of workflows in their workshop, primarily utilizing Python. Lastly, even if this might not be a programming role in the traditional sense, I am currently working as a student tutor for two programming courses at university, as which I review C++ assignments of student and hold off weekly tutorial session to support students in learning fundamental C++ and Java concepts, as well as the basics of the C++ STL.

One motivation for wanting to contribute to HPX is to enhance my proficiency in C++, especially as I wish to learn more about designing and implementing user-friendly, but performant C++ libraries, while also deepening my knowledge on leveraging advanced techniques, like template metaprogramming, for this purpose. Besides this, another reason is my interest in parallel computing, particularly in the various approaches to abstracting parallelism and asynchrony into intuitive concepts, aiming to simplify their application and to reduce potential errors.

Working on the S/R support of HPX would align well with both of these interests, as this project presents not only an opportunity to gain insights into the core development of a complex C++ library like HPX, but also to explore senders and receivers more deeply, which I find both compelling and captivating models for asynchronous programming.

Beyond the Google Summer of Code, I would be eager to maintain and polish the developed solution, as well as to help with adapting it to possible changes in the underlying standard proposals and other further developments, such as HPX specific extensions.

I would rate my knowledge of the selected technologies on a scale from 0 to 5 as follows:

C++	3
C++ Standard Library	3
Boost C++ Libraries	1
Git	3

The repository demonstrating a simple matrix multiplication with HPX can be found here: github.com/zhekemist/hpx-matrix-multiplication

In terms of software development tools, I primarily use the IDEs by JetBrains, particularly CLion for C++. I also have some experience with Eclipse. As for documentation tools, I am familiar with Sphinx, mainly for documenting Python code.