Performance analysis and optimization on HPX Rotate algorithm

Chuanqiu He , Panagiotis Syskakis, Hartmut Kaiser
Center for Computation and Technology, Louisiana State University, Baton Rouge, LA 70803, USA

Abstract

HPX is a C++ Standard Library for Parallelism and Concurrency, which implements parallel versions of all C++ Standard algorithms.
This poster presents the optimization of the HPX parallel rotate algorithm. The original implementation failed to effectively distribute work among CPU cores and caused un-needed overhead due to redundant usage of a hpx::future. **The optimization consists of introducing core partitioning functionality and removing unnecessary overhead**.
The results show significant performance gain, averaging a 2x improvement for input sizes smaller than 2^22 compared to the original implementation.

Introduction

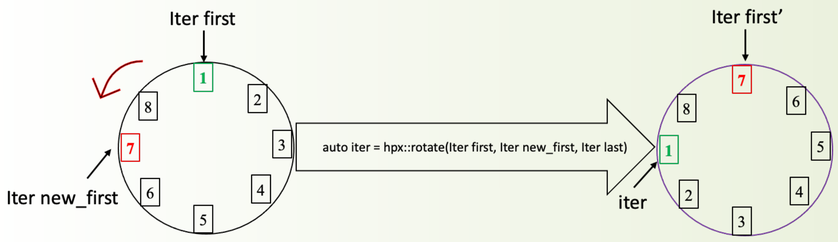
What is HPX

HPX is an open-source C++ library for concurrency and parallelism.
It provides all corresponding facilities included in the C++ Standard, including parallel implementations for all algorithms in the C++ Standard Template Library. HPX also implements (and extends) proposals yet to be included in the C++ Standard.

Overall, HPX provides much needed tools for developing parallel and asynchronous applications. The hpx::future (like std::future) is a lightweight mechanism to chain asynchronous operations and access their results when ready. HPX also provides high-level directives for managing computing resources. The programmer can control CPU affinity, limit execution on selected CPU cores, control priority of asynchronous tasks, and more.

How HPX parallel rotate works

```
template <typename ExPolicy,  typename Iter>
Iter hpx::rotate(ExPolicy && p, Iter first, Iter new_first, Iter last)
```



- Performs a left rotation on a range of elements. Specifically, rotate swaps the elements in the range [first, last) in such a way that the element ***new_first*** becomes the first element of the new range and new_first - 1 becomes the last element.
- The rotate algorithm **returns** a hpx::future<Iter> if the execution policy is of type sequenced or parallel task policy and returns Iter otherwise. The rotate algorithm returns the iterator equal to first + (last - new_first).

Implementation of HPX parallel rotate

The complexity of this algorithm is O(n), and HPX provides an elegant parallel implementation, which achieves the desired result using three reverse operations.

The **new_first** iterator separates the starting vector into two sections. Thus, we proceed as follows (Fig. 1):

Step 1: Reverse the left and right sections of the vector with full number of cores separately.
Step 2: Reverse the whole vector.

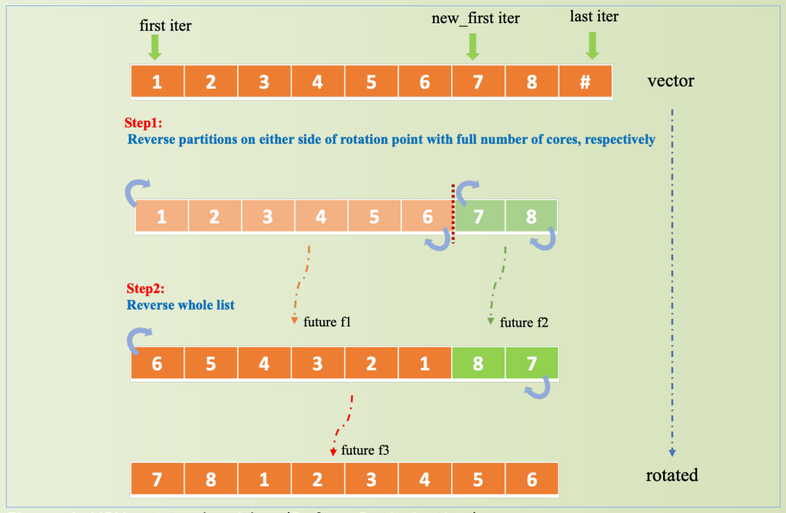


Figure 1: HPX rotate algorithm (Before Optimization)

Optimization

Motivation

The existing implementation of HPX rotate invokes reverse on left and right section with full number of cores, separately.

Bottlenecks:

- The two reverse operations in step 1, while scheduled concurrently, most likely didn't run at the same time. That's because all cores used for the one reverse operation didn't allow for the concurrent reverse operation to go on, as both were scheduled to run on the same set of cores.
- An additional unnecessary asynchronous HPX task (hpx::future) was created for step 2.

Improving performance methods

Add core partition functionality

✓ The number of cores allocated to reversing the left and right sections were selected proportionally to each section's size.
✓ This enables the two reverse operations to be executed concurrently.
✓ Proper placement of work-chunks to cores was added to facilitate the above change. This eliminated overlapping scheduling and reduced contention.

Remove unnecessary future f3

✓ The additional future in step 2 was eliminated, avoiding unnecessary overhead.

❑ The optimizations were applied as follows (Fig. 2):

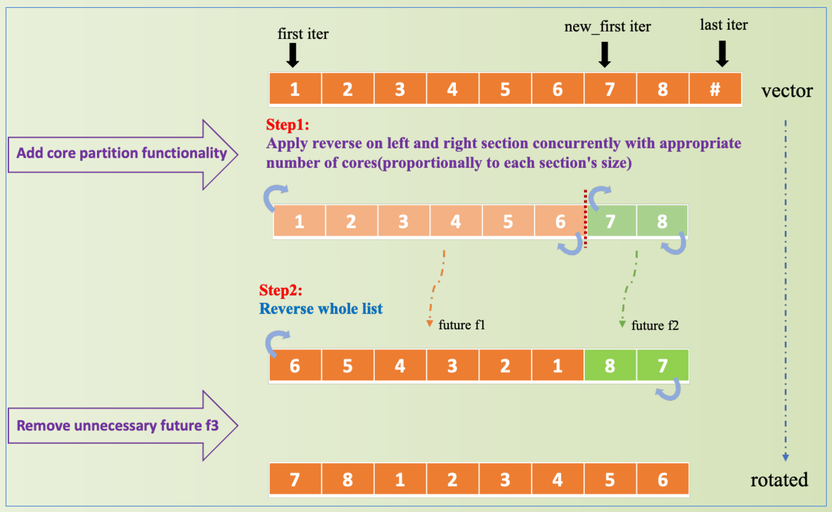


Figure 2: HPX rotate algorithm (After Optimization)

Results

Improvement of rotate algorithm after core partitioning

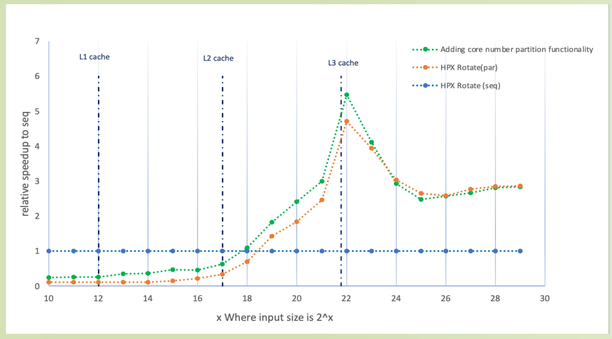


Figure 3: Performance of HPX rotate after adding core partition functionality. Number of threads = 8

Improvement of rotate algorithm after removing future f3

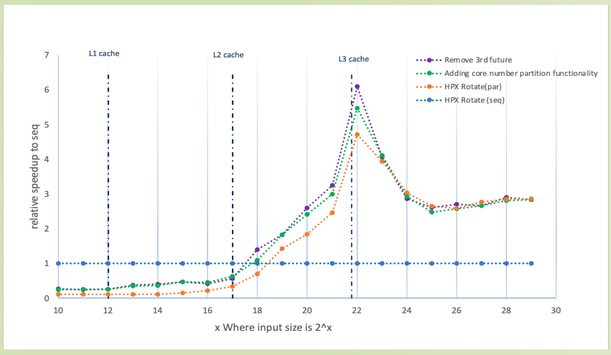


Figure 4: Performance of HPX rotate after removing unnecessary future and adding core partition functionality. Number of threads = 8

Strong Scaling

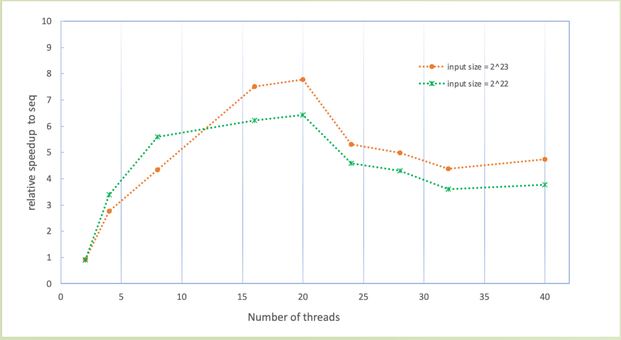


Figure 5: Strong Scaling of HPX Rotate Algorithm

Conclusions

- The core partitioning accounts for most of the performance gain, as evident by the performance graphs. The improvement is more significant on smaller input sizes (smaller than 2^22), showing 2x gain on average.
- This can be accredited to the fewer (thus larger) data chunks compared to using all cores for both sections of the vector. Thus, fewer asynchronous tasks are spawned, reducing the corresponding overhead. Additionally, the two reverse operations are executed concurrently with less contention.
- Removing the unnecessary future gave a small additional performance boost, achieving 11% improvement at peak points.

References

- Kaiser, Hartmut, et al. "Hpx-the c++ standard library for parallelism and concurrency." *Journal of Open Source Software* 5.53 (2020): 2352.
- Alexander Stepanov. *Notes on programming*. 2018. http://stepanovpapers.com/notes.pdf