CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

HPX and Kokkos: unifying asynchrony and portability on the path towards standardization

WAMTA23

Mikael Simberg (Swiss National Supercomputing Centre), Gregor Daiß (University of Stuttgart)

February 17th, 2023

std::execution

`std::execution`: what is it?

- **Generic framework for asynchrony**
  - Integrates and replaces previous proposals (most notably P0443)
- Considered for inclusion in C++26
- Handles to execution contexts: **schedulers** (previously executors)
- Handles to asynchronous values: **senders** (previously futures)
- **Algorithms** for adapting, combining, and consuming senders
  - Allow building the DAG of work
- Interoperates with coroutines
- "sender/receiver" is the same as `std::execution`

`std::execution`: example

```
sender = std::move(sender) |
      ex::transfer(ex::with_stacksize(ex::thread_pool_scheduler{}, stacksize)) |
      ex::then(std::move(f_setup)) |
      ex::bulk(n, std::move(f)) |
      ex::then(std::move(f_finalize)) |
      ex::ensure_started();
```

Note: transfer to hopefully be replaced by a scoped on algorithm in the future

`std::execution`: why?

- **Customization** and **zero-overhead**
  - std::future type-erased, leaves little room for customization
  - std::execution decomposes work description and submission into low-level basis operations: allows eliding heap allocations in many situations
- std::execution is low level for those that need it, **surface syntax is simple for users**
- **Interoperability between different libraries**
- More information
  - Working with Asynchrony Generically: A Tour of C++ Executors (Eric Niebler)
    - https://youtube.com/watch?v=xLboNIf7BTg
    - https://youtube.com/watch?v=6a0zzUBUNW4
  - https://wg21.link/p2300

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

HPX and Kokkos

HPX and Kokkos

HPX

- **Lightweight CPU tasking runtime**
- Interoperability with asynchronous APIs of CUDA, HIP, SYCL (in progress), MPI
- Implements previous and current C++ proposals
- Full implementation of C++ parallel algorithms (including ranges)
- **Involved in C++ standardization**

Kokkos

- **Performance portability layer**
- Portable execution and memory management on all major runtimes/programming models
- Full implementation of C++ parallel algorithms
- **Involved in C++ standardization**

HPX and Kokkos: previous work

- HPX backend in Kokkos
  - Built on HPX futures, executors
- HPX-Kokkos interoperability layer
  - Futures from some Kokkos backends (HPX, CUDA, HIP, SYCL in progress)
- Used in Octo-Tiger

G. Daiß, S. Y. Singanaboina, P. Diehl, H. Kaiser and D. Pflüger, "From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types," 2022 IEEE/ACM 7th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), Dallas, TX, USA, 2022, pp. 10-19, doi: 10.1109/ESPM256814.2022.00007.

G. Daiß et al., "Beyond Fork-Join: Integration of Performance Portable Kokkos Kernels with HPX," 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Portland, OR, USA, 2021, pp. 377-386, doi: 10.1109/IPDPSW52791.2021.00066.

HPX and Kokkos: this work

- **Validate usability and performance of std::execution**
- `std::execution` in HPX
  - Implements the majority of std::execution using C++17
  - Eventually replace by reference implementation (https://github.com/NVIDIA/stdexec) or standard library experimental implementations
- HPX's `std::execution` implementation for Kokkos backend
  - Almost no visible API changes; added way to get a sender from instance

```
sender = std::move(sender) |
      ex::transfer(ex::with_stacksize(ex::thread_pool_scheduler{}, stacksize)) |
      ex::then(std::move(f_setup)) |
      ex::bulk(n, std::move(f)) |
      ex::then(std::move(f_finalize)) |
      ex::ensure_started();
```

`std::execution` experience in HPX and Kokkos

The good

- `bulk`, `then`, `transfer` are sufficient algorithms to implement Kokkos backend
- Performance same or better compared to previous implementation
- Customization and `tag_invoke`, powerful
- Straightforward generalization from futures to senders and executors to schedulers, makes transition easier

The bad

- Compilation times and bloat (but `std::execution` is not unique)
- No type-erased sender (but this is planned as an extension)

The ugly

- `tag_invoke`

Unknown

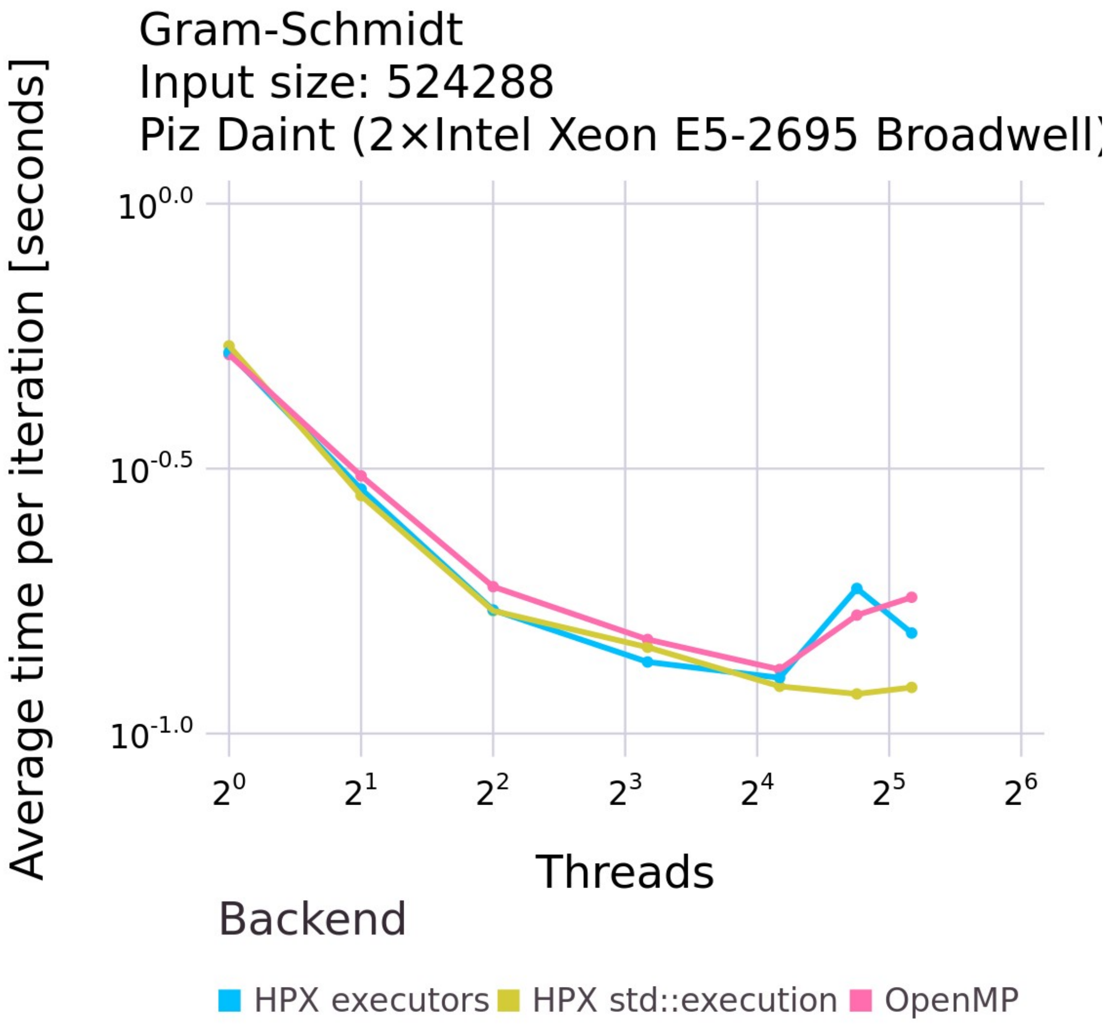
- Memory management not part of `std::execution`: will there be something or will we all use unified memory by then?

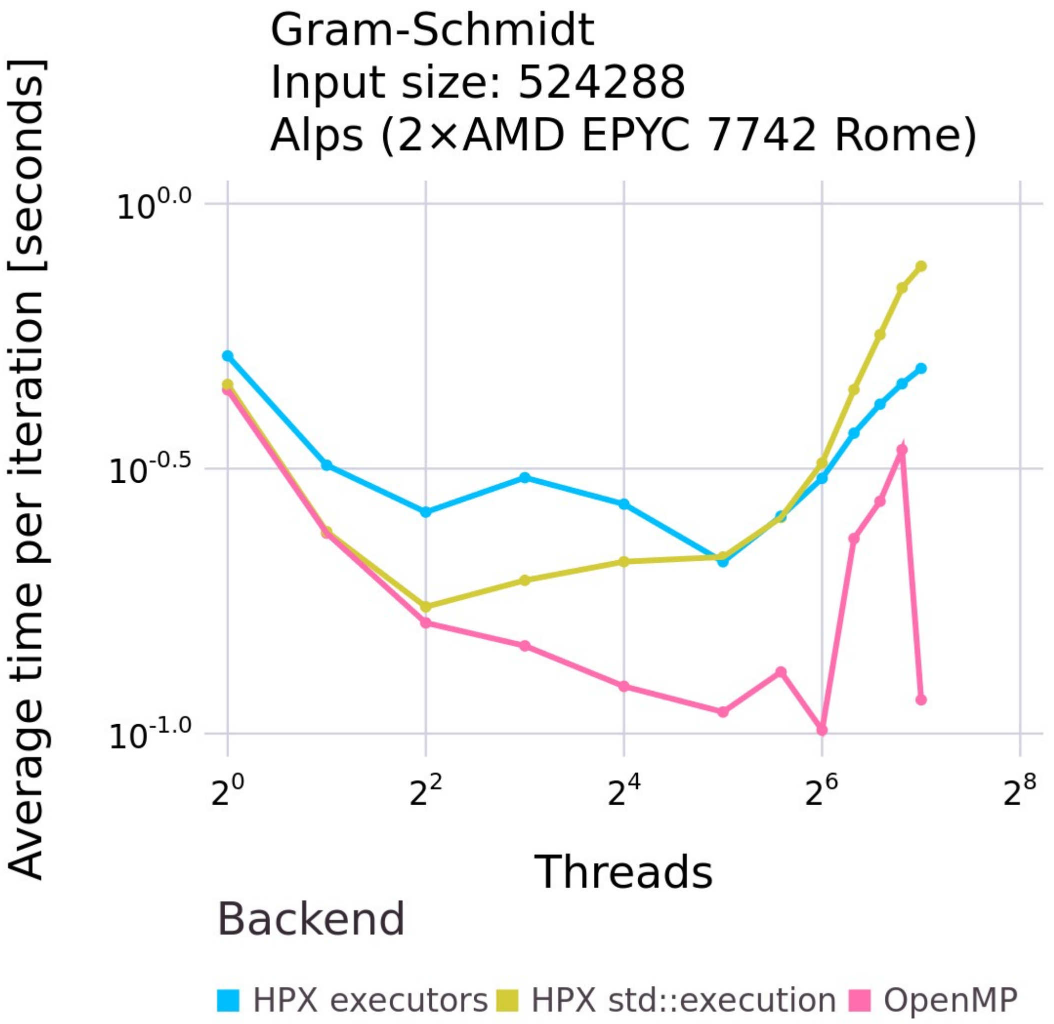
Benchmarks

Benchmarks

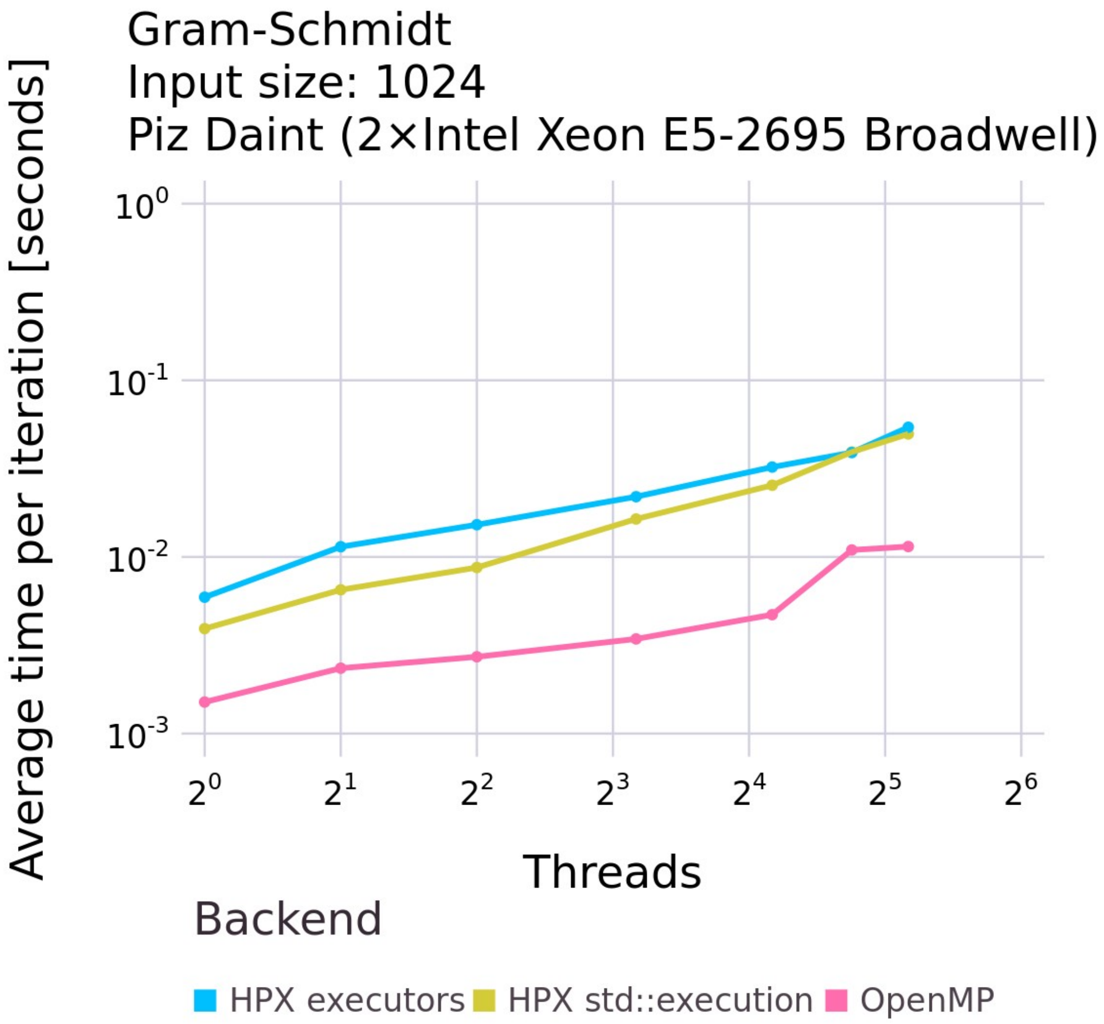
- Kokkos Gram-Schmidt performance test
  - Not full application, but gives a good indication about relative performance
  - Fork-join with for loops and reductions
- Octo-Tiger gravity-only scenario (three levels)
  - Simulation of binary star mergers
  - Octree, many independent kernels created while traversing tree
  - Monopole and multipole kernels can run with a single task or many tasks
  - See later presentation!
- Preliminary results, not much effort has been put into optimizations

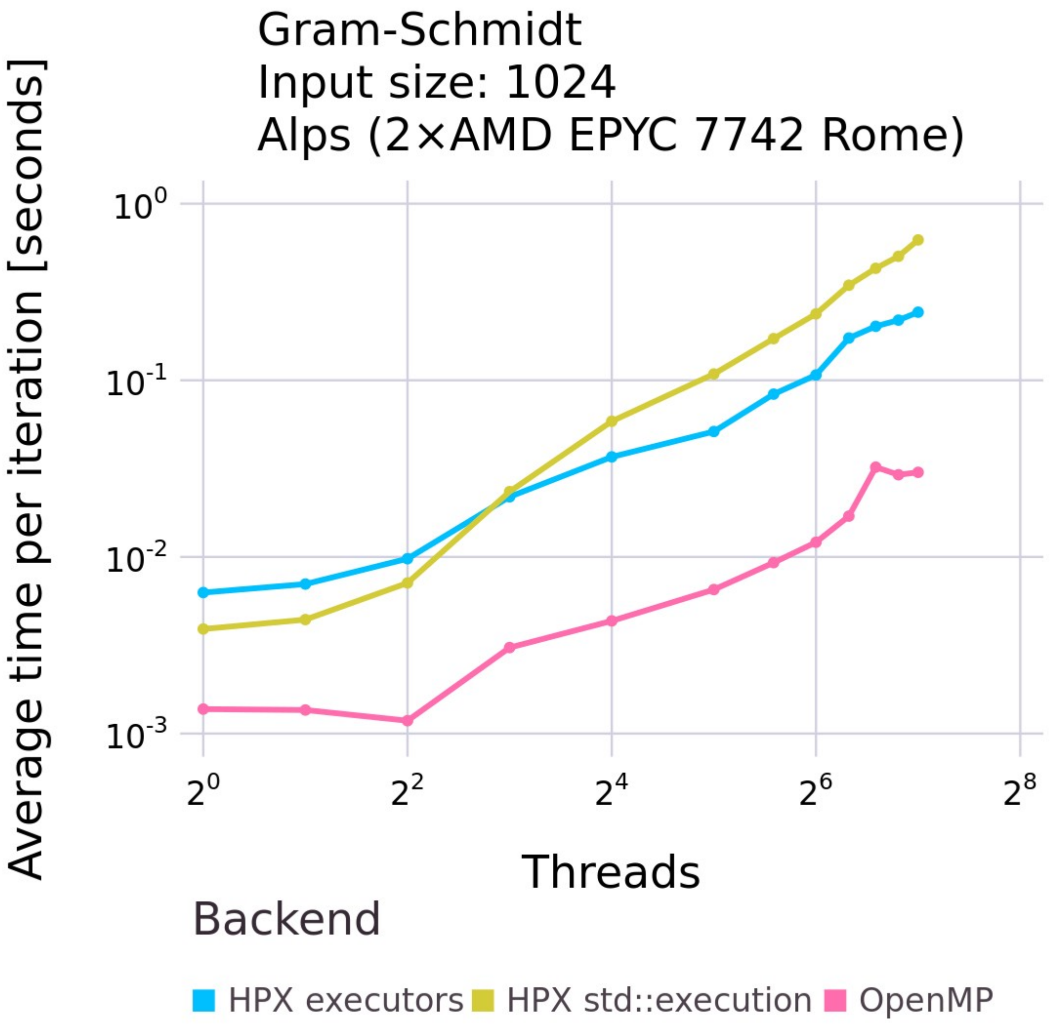
Benchmarks: Kokkos Gram-Schmidt



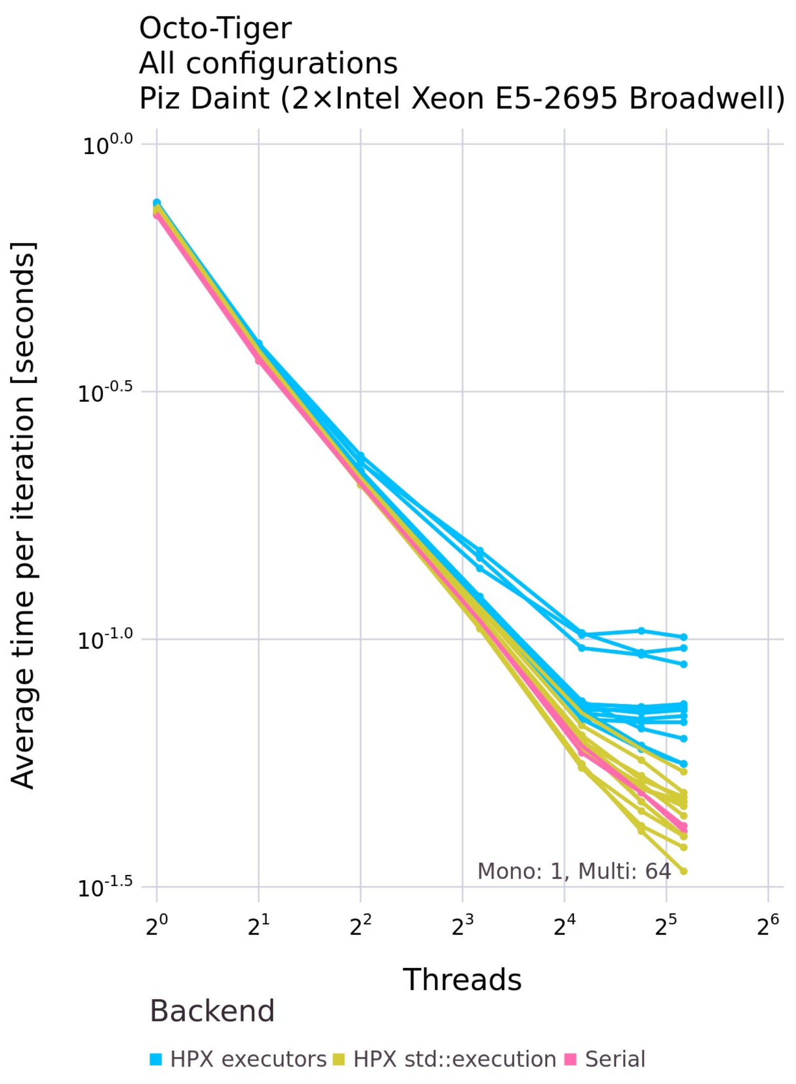


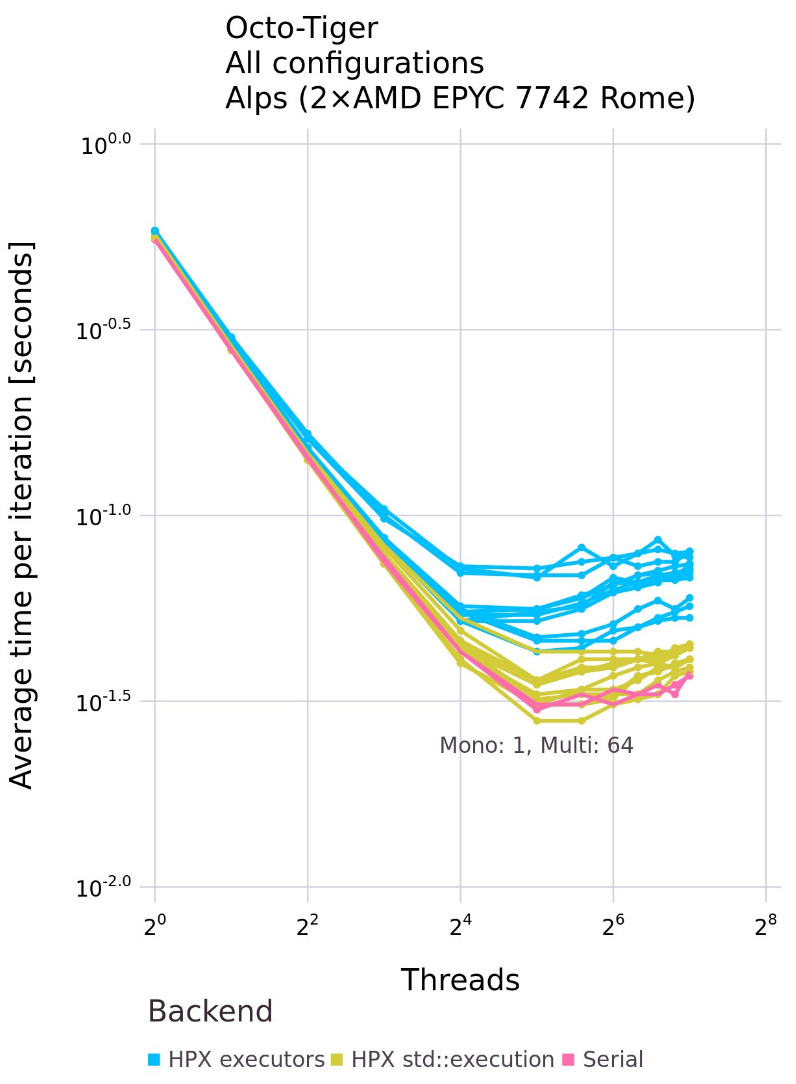
Benchmarks: Kokkos Gram-Schmidt

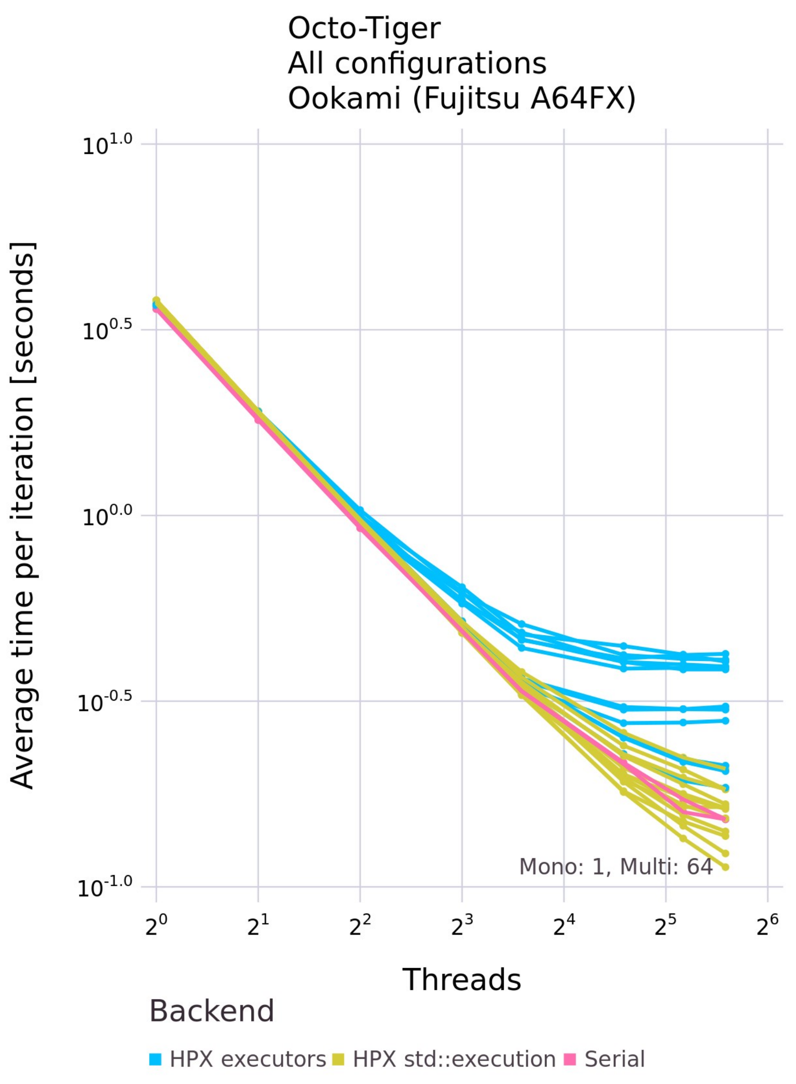




Benchmarks: Octo-Tiger gravity-only







Benchmarks

- Performance *generally* the same or better with std::execution backend
- Not solely thanks to std::execution, but it does help
  - Example of std::execution improvement: bulk operations don’t need one future per task, can combine them into one bigger allocation in the operation state
  - Example of std::execution improvement: lazy construction of DAG means that many internal locks required by futures are no longer required
  - Example of non-std::execution improvement: spawning only one task per worker thread and running a “mini-scheduler” on them for a parallel region (though this can also be slower in some situations)

Conclusion

Outlook and future work

- Can e.g. Kokkos support any std::execution scheduler? How much customization is required to make it work? How much customization is required to make it fast?
- Can all the C++ parallel algorithms be written on top of std::execution? All signs point to yes, but we haven't done that yet. Same concerns as above.
- std::execution gives interoperability between most runtimes, but contention between CPU thread pools is still a problem.
- What should asynchronous parallel algorithms look like?
- What should communication/remote execution look like? MPI? Lower-level libraries like libfabric?

Conclusion

- Standardization is an important step to collect knowledge that has accumulated in separate libraries and communities
  - Combining HPX, Kokkos, std::execution is one step of validating std::execution (in our view, a successful step)
- std::execution gives a more generic framework with the same or better performance as HPX’s futures and executors
- **Great time to start trying out std::execution**

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 $5M National Science Foundation grant (#1927880).

Benchmark details

- Kokkos commits
  - HPX `std::execution`: `5ea96bca`
  - HPX executors/OpenMP/Serial: `879d6079`
- HPX: `d09db415`
  - Networking off
  - Jemalloc
- HPX-Kokkos: `3383f78a`
- Octo-Tiger: `3d3511f4`
  - With default SIMD support
- Compilers:
  - Piz Daint: Cray CCE 12
  - Alps: GCC 11
  - Ookami: GCC 12