The Journal of Open Source Software

HPX - The C++ Standard Library for Parallelism and Concurrency

Parsa Amini[1], **Bryce Adelstein Lelbach**[5], **Agustín Berge**[6], **John Biddiscombe**[4], **Steven R. Brandt**[1], **Patrick Diehl**[1], **Nikunj Gupta**[3], **Thomas Heller**[2], **Kevin Huck**[8], **Hartmut Kaiser**[1], **Zahra Khatami**[7], **Alireza Kheirkhahan**[1], **Adrian S. Lemoine**[6], **Auriane Reverdell**[4], **Shahrzad Shirzad**[1], **Mikael Simberg**[4], **Bibek Wagle**[1], **Weile Wei**[1], and **Tianyi Zhang**[6]

1 Center for Computation & Technology, Louisiana State University, LA, Baton Rouge, United States of America **2** Exasol, Erlangen, Germany **3** Indian Institute of Technology, Roorkee, India **4** Swiss National Supercomputing Centre, Lugano, Switzerland **5** NVIDIA, CA, Santa Clara, United States of America **6** STE||AR Group **7** Oracle, CA, Redwood City, United States of America **8** Oregon Advanced Computing Institute for Science and Society (OACISS), University of Oregon, OR, Eugene, United States of America

DOI: 10.21105/joss.02352

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @bhatele
- @davidbeckingsale

Submitted: 12 June 2020
Published: 30 July 2020

Summary

The new challenges presented by Exascale system architectures have resulted in difficulty achieving a desired scalability using traditional distributed-memory runtimes. Asynchronous many-task systems (AMT) are based on a new paradigm showing promises in addressing these challenges, providing application developers with a productive and performant approach to programming on next generation systems.

A detailed comparison of various AMT's is given in (Thoman et al., 2018). Some notable AMT solutions are: Uintah (Germain, McCorquodale, Parker, & Johnson, 2000), Chapel (Chamberlain, Callahan, & Zima, 2007), Charm++ (Kale & Krishnan, 1993), Kokkos (Edwards, Trott, & Sunderland, 2014), Legion (Bauer, Treichler, Slaughter, & Aiken, 2012), and PaRSEC (Bosilca et al., 2013). Note that we only refer to distributed memory solutions, since this is one important feature for scientific applications to run large scale simulations. The major show piece of HPX compared to the mentioned distributed AMTs is its future-proof C++ standard conform-ing API.

HPX is a C++ Library for Concurrency and Parallelism that is developed by The STE||AR Group, an international group of collaborators working in the field of distributed and parallel programming (Heller, Diehl, Byerly, Biddiscombe, & Kaiser, 2017; Kaiser et al., n.d.; Tabbal, Anderson, Brodowicz, Kaiser, & Sterling, 2011). It is a runtime system written using modern C++ techniques that is linked as part of an application. HPX exposes extended services and functionalities supporting the implementation of parallel, concurrent, and distributed capabilities for applications in any domain - it has been used in scientific computing, gaming, finances, data mining, and other fields.

HPX's main goal is to improve efficiency and scalability of parallel applications by increasing resource utilization and reducing synchronization through providing an asynchronous API and employing adaptive scheduling. The consequent use of *Futures* intrinsically enables overlap of computation and communication and constraint-based synchronization. HPX is able to maintain a balanced load among all the available resources resulting in significantly reducing processor starvation and effective latencies while controlling overheads. HPX is fully conforming to the C++ ISO Standards and implements the standardized concurrency mechanisms

The Journal of Open Source Software

and parallelism facilities. Further, HPX extends those facilities to distributed use cases, thus enabling syntactic and semantic equivalence of local and remote operations on the API level. HPX uses the concept of C++ *Futures* to transform sequential algorithms into wait-free asynchronous executions. The use of *Futurization* enables the automatic creation of dynamic data flow execution trees of potentially millions of lightweight HPX tasks executed in the proper order. HPX also provides a work-stealing task scheduler that takes care of fine-grained parallelizations and automatic load balancing. Furthermore, HPX implements functionalities proposed as part of the ongoing C++ standardization process.

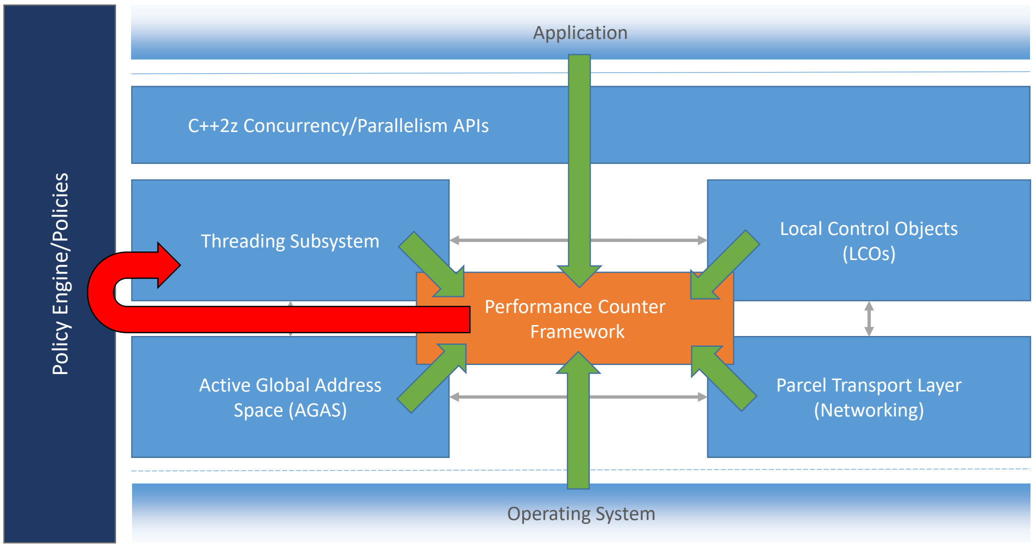

Figure 1: Sketch of HPX's architecture with all the components and their interactions.

Figure 1 sketches HPX's architectures. The components of HPX and their references are listed below:

- **Threading Subsystem** (Kaiser, Brodowicz, & Sterling, 2009) The thread manager manages the light-weight user level threads created by HPX. These light-weight threads have extremely short context switching times resulting in the reduced latencies even for very short operations. HPX provides following pre-defined scheduling policies: static, thread local, and hierarchical.
- **Active Global Address Space (AGAS)** (Amini & Kaiser, 2019; Kaiser, Heller, Adelstein-Lelbach, Serio, & Fey, 2014) To support distributed objects, HPX supports a global address resolution component that is extending the PGAS model to enable runtime based resource allocation and data placement. This layer enables HPX to expose a uniform API for local and remote execution. Unlike PGAS, AGAS provides the user with the ability to transparently move global objects in a distributed system. This enables AGAS to support load balancing via object migration.
- **Parcel Transport Layer** (Biddiscombe, Heller, Bikineev, & Kaiser, 2017; Kaiser et al., 2009) This component is an active-message networking layer. The parcelport is able to leverage AGAS in order to launch functions on global objects regardless of their current placement in a distributed system. Additionally its asynchronous protocol enables the parcelport to implicitly overlap communication and computation. The parcelport is modular to support multiple communication library backends. By default HPX supports TCP/IP, Message passing Interface (MPI), and libfabric (Daiß et al., 2019).
- **Performance counters** (Grubel, 2016) HPX provides its users with a uniform suite of performance counters to monitor system metrics that are accessible globally. These counters have their names registered with AGAS, which enables the users to easily

query for different metrics at runtime. Additionally, HPX provides an API for users to create their own counters to gather information customized to their own application. By default HPX provides performance counters for its components, such as networking, AGAS operations, thread scheduling, and various statistics.

- **Policy Engine/Policies** (Huck et al., 2015; Khatami, Troska, Kaiser, Ramanujam, & Serio, 2017; Laberge et al., 2019) Often, modern applications must adapt to runtime environments to ensure acceptable performance. Autonomic Performance Environment for Exascale (APEX) enables this flexibility by measuring HPX tasks, monitoring system utilization, and accepting user provided policies that are triggered by defined events. In this way, features such as parcel coalescing (Wagle, Kellar, Serio, & Kaiser, 2018) can adapt to the current phase of an application or even state of a system.
- **Accelerator Support** HPX has support for two methods of integration with GPUs: HPXCL (Diehl et al., 2018b; Stumpf et al., 2018) and HPX.Compute (Copik & Kaiser, 2017) HPXCL provides users the ability to manage GPU kernels through a global object. This enables HPX to coordinate the launching and synchronization of CPU and GPU code. HPX.Compute (Copik & Kaiser, 2017) aims to provide a single source solution to heterogeneity by automatically generating GPU kernels from C++ code. This enables HPX to launch both CPU and GPU kernels as dictated by the current state of the system.
- **Local Control Objects** HPX has support for many of the C++20 primitives, such as `hpx::latch`, `hpx::barrier`, and `hpx::counting_semaphore` to synchronize the code or overlap computation and communication. These functions are standard conform according to the C++20 (ISO/IEC, 2020). For asynchronous computing HPX provides `hpx::async` and `hpx::future`, see the second example in the next section.
- **Software Resilience** HPX supports software level resilience (Gupta, Mayo, Lemoine, & Kaiser, 2020) through its resiliency API, such as `hpx::async_replay` and `hpx::async_replicate` and its dataflow counterparts `hpx::dataflow_replay` and `hpx::dataflow_replicate`. These APIs are resilient against memory bit flips and other hardware errors. HPX provides an easy method to port codes to the resilient API by replacing `hpx::async` or `hpx::dataflow` with its resilient API counterparts everywhere in the code without making any other changes.
- **C++ Standards conforming API** HPX implements all of the C++17 parallel algorithms (ISO/IEC, 2017) and extends those with asynchronous versions. Here, HPX provides the `hpx::execution::seq`, `hpx::execution::par` execution policies, and (as an extension) their asynchronous equivalents `hpx::execution::seq(hpx::execution::task)` and `hpx::execution::par(hpx::execution::task)` (see the first code example in the next section). HPX also implements the C++20 concurrency facilities and APIs (ISO/IEC, 2020), such as `hpx::jthread`, `hpx::latch`, `hpx::barrier`, etc.

HPX is utilized in a diverse set of applications: Octo-Tiger (Daiß et al., 2019; Heller et al., 2019; Pfander, Daiß, Marcello, Kaiser, & Pflüger, 2018), an astrophysics code for stellar mergers; libGeoDecomp (Schäfer & Fey, 2008), an auto-parallelizing library to speed up stencil code based computer simulations; NLMech (Diehl et al., 2018a), a simulation tool for non-local models, e.g. Peridynamics; hpxMP (Zhang et al., 2019, 2020); Kokkos, C++ Performance Portability Programming EcoSystem (Carter Edwards, Trott, & Sunderland, 2014); Dynamical Cluster Approximation (DCA++), a high-performance research software framework to solve quantum many-body problems with cutting edge quantum cluster algorithms (Hähner et al., 2020); a modern OpenMP implementation leveraging HPX that supports shared memory multithread programming; and Phylanx (Tohid et al., 2018; Wagle et al., 2019) a distributed array toolkit.

Example code

The following is an example of HPX's parallel algorithms API using execution policies as defined in the C++17 Standard (ISO/IEC, 2017). HPX implements all of the parallel algorithms defined therein. The parallel algorithms extend the classic STL algorithms by adding an additional first argument (called execution policy). The `hpx::execution::seq` implies sequential execution while `hpx::execution::par` will execute the algorithm in parallel. HPX's parallel algorithm library API is completely standards conforming.

```
#include <hpx/hpx.hpp>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> values = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Compute the sum in a sequential fashion
    int sum1 = hpx::reduce(
        hpx::execution::seq, values.begin(), values.end(), 0);
    std::cout << sum1 << '\n';      // will print 55

    // Compute the sum in a parallel fashion based on a range of values
    int sum2 = hpx::reduce(hpx::execution::par, values, 0);
    std::cout << sum2 << '\n';      // will print 55 as well

    return 0;
}
```

Example for the HPX's concurrency API where the Taylor series for the $\sin(x)$ function is computed. The Taylor series is given by

$$\sin(x) \approx= \sum_{n=0}^{N} (-1)^{n-1} \frac{x^{2n}}{(2n)!}.$$

For the concurrent computation, the interval $[0, N]$ is split in two partitions from $[0, N/2]$ and $[(N/2) + 1, N]$ and these are computed asynchronously using `hpx::async`. Note that each asynchronous function call returns an `hpx::future` which is needed to synchronize the collection of the partial results. The future has a `get()` method that returns the result once the computation of the Taylor function finished. If the result is not ready yet, the current thread is suspended until the result is ready. Only if `f1` and `f2` are ready, the overall result will be printed to the standard output stream.

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

// Define the partial taylor function
double taylor(size_t begin, size_t end, size_t n, double x)
{
    double denom = factorial(2 * n);
    double res = 0;
    for (size_t i = begin; i != end; ++i)
```

```
    {
        res += std::pow(-1, i - 1) * std::pow(x, 2 * n) / denom;
    }
    return res;
}

int main()
{
    // Compute the Talor series sin(2.0) for 100 iterations
    size_t n = 100;

    // Launch two concurrent computations of each partial result
    hpx::future<double> f1 = std::async(taylor, 0, n / 2, n, 2.);
    hpx::future<double> f2 = std::async(taylor, (n / 2) + 1, n, n, 2.);

    // Introduce a barrier to gather the results
    double res = f1.get() + f2.get();

    // Print the result
    std::cout << "Sin(2.) = " << res << std::endl;
}
```

Please report any bugs or feature requests on the HPX's GitHub page.

Acknowledgments

We would like to acknowledge the National Science Foundation (NSF), the U.S. Department of Energy (DoE), the Defense Technical Information Center (DTIC), the Defense Advanced Research Projects Agency (DARPA), the Center for Computation and Technology (CCT) at Louisiana State University (LSU), the Swiss National Supercomputing Centre (CSCS), the Department of Computer Science 3 - Computer Architecture at the University of Erlangen Nuremberg who fund and support our work, and the Heterogeneous System Architecture (HSA) Foundation.

We would also like to thank the following organizations for granting us allocations of their compute resources: LSU HPC, Louisiana Optical Network Iniative (LONI), the Extreme Science and Engineering Discovery Environment (XSEDE), the National Energy Research Scientific Computing Center (NERSC), the Oak Ridge Leadership Computing Facility (OLCF), Swiss National Supercomputing Centre (CSCS/ETHZ), the Juelich Supercomputing Centre (JSC), and the Gauss Center for Supercomputing.

At the time the paper was written, HPX was directly funded by the following grants:

- The National Science Foundation through awards 1339782 (STORM) and 1737785 (Phylanx).
- The Department of Energy (DoE) through the awards DE-AC52-06NA25396 (FLeCSI) DE-NA0003525 (Resilience), and DE-AC05-00OR22725 (DCA++).
- The Defense Technical Information Center (DTIC) under contract FA8075-14-D-0002/0007.
- The Bavarian Research Foundation (Bayerische Forschungsstiftung) through the grant AZ-987-11.

- The European Commission's Horizon 2020 programme through the grant H2020-EU.1.2.2. 671603 (AllScale).

For a constantly updated list of previous and current funding, we refer to the corresponding HPX's website.

References

Amini, P., & Kaiser, H. (2019). Assessing the Performance Impact of using an Active Global Address Space in HPX: A Case for AGAS. In *2019 ieee/acm third annual workshop on emerging parallel and distributed runtime systems and middleware (ipdrm)* (pp. 26–33). doi:10.1109/ipdrm49579.2019.00008

Bauer, M., Treichler, S., Slaughter, E., & Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the international conference on high performance computing, networking, storage and analysis* (pp. 1–11). IEEE. doi:10.1109/SC.2012.71

Biddiscombe, J., Heller, T., Bikineev, A., & Kaiser, H. (2017). Zero Copy Serialization using RMA in the Distributed Task-Based HPX Runtime. In *14th international conference on applied computing*. IADIS, International Association for Development of the Information Society.

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., & Dongarra, J. J. (2013). Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, *15*(6), 36–45. doi:10.1109/MCSE.2013.98

Carter Edwards, H., Trott, C. R., & Sunderland, D. (2014). Kokkos. *J. Parallel Distrib. Comput.*, *74*(12), 3202?3216. doi:10.1016/j.jpdc.2014.07.003

Chamberlain, B. L., Callahan, D., & Zima, H. P. (2007). Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, *21*(3), 291–312. doi:10.1177/1094342007078442

Copik, M., & Kaiser, H. (2017). Using SYCL as an implementation framework for hpx. Compute. In *Proceedings of the 5th international workshop on opencl* (pp. 1–7). doi:10.1145/3078155.3078187

Daiß, G., Amini, P., Biddiscombe, J., Diehl, P., Frank, J., Huck, K., Kaiser, H., et al. (2019). From Piz-Daint to the Stars: Simulation of Stellar Mergers using High-level abstractions. In *Proceedings of the international conference for high performance computing, networking, storage and analysis* (pp. 1–37). doi:10.1177/1094342018819744

Diehl, P., Jha, P. K., Kaiser, H., Lipton, R., & Levesque, M. (2018a). Implementation of Peridynamics utilizing HPX–the C++ Standard Library for Parallelism and Concurrency. *arXiv preprint arXiv:1806.06917*.

Diehl, P., Seshadri, M., Heller, T., & Kaiser, H. (2018b). Integration of CUDA Processing within the C++ Library for Parallelism and Concurrency (HPX). In *2018 ieee/acm 4th international workshop on extreme scale programming models and middleware (espm2)* (pp. 19–28). IEEE. doi:10.1109/espm2.2018.00006

Edwards, H. C., Trott, C. R., & Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, *74*(12), 3202–3216. doi:10.1016/j.jpdc.2014.07.003

Germain, J. D. de S., McCorquodale, J., Parker, S. G., & Johnson, C. R. (2000). Uintah: A massively parallel problem solving environment. In *Proceedings the ninth international* *symposium on high-performance distributed computing* (pp. 33–41). IEEE. doi:10.1109/HPDC.2000.868632

Grubel, P. A. (2016). *Dynamic Adaptation in HPX: A Task-based Parallel Runtime System* (PhD thesis). New Mexico State University.

Gupta, N., Mayo, J. R., Lemoine, A. S., & Kaiser, H. (2020). Implementing Software Resiliency in HPX for Extreme Scale Computing. doi:10.2172/1614897

Hähner, U. R., Alvarez, G., Maier, T. A., Solcà, R., Staar, P., Summers, M. S., & Schulthess, T. C. (2020). DCA++: A software framework to solve correlated electron problems with modern quantum cluster methods. *Computer Physics Communications*, *246*, 106709. doi:10.1016/j.cpc.2019.01.006

Heller, T., Diehl, P., Byerly, Z., Biddiscombe, J., & Kaiser, H. (2017). HPX–An Open Source C++ Standard Library for Parallelism and Concurrency. *Proceedings of OpenSuCo*, 5.

Heller, T., Lelbach, B. A., Huck, K. A., Biddiscombe, J., Grubel, P., Koniges, A. E., Kretz, M., et al. (2019). Harnessing Billions of Tasks for a Scalable Portable Hydrodynamic Simulation of the Merger of two Stars. *The International Journal of High Performance Computing Applications*, *33*(4), 699–715. doi:10.1177/1094342018819744

Huck, K. A., Porterfield, A., Chaimov, N., Kaiser, H., Malony, A. D., Sterling, T., & Fowler, R. (2015). An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, *2*(3), 49–66. doi:10.14529/jsfi150305

ISO/IEC, S. (2017). ISO International Standard ISO/IEC 14882:2017(E) - Programming Language C++. *Geneva, Switzerland: International Organization for Standardization (ISO)*.

ISO/IEC, S. (2020). ISO International Standard ISO/IEC 14882:2020(E) - Programming Language C++. [Working draft]. *Geneva, Switzerland: International Organization for Standardization (ISO)*.

Kaiser, H., Brodowicz, M., & Sterling, T. (2009). ParalleX: An Advanced Parallel Execution Model for Scaling-impaired Applications. In *2009 international conference on parallel processing workshops* (pp. 394–401). IEEE. doi:10.1109/icppw.2009.14

Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., & Fey, D. (2014). HPX: A Task based Programming Model in a Global Address Space. In *Proceedings of the 8th international conference on partitioned global address space programming models* (pp. 1–11).

Kaiser, H., Heller, T., Simberg, M., Berge, A., Biddiscombe, J., Reverdell, A., Huck, K., et al. (n.d.). *HPX: The C++ Standards Library for Parallelism and Concurrency*. *GitHub repository*. Zenodo. doi:10.5281/zenodo.598202

Kale, L. V., & Krishnan, S. (1993). Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on object-oriented programming systems, languages, and applications* (pp. 91–108).

Khatami, Z., Troska, L., Kaiser, H., Ramanujam, J., & Serio, A. (2017). HPX Smart Executors. In *Proceedings of the third international workshop on extreme scale programming models and middleware*, ESPM2'17. New York, NY, USA: Association for Computing Machinery. doi:10.1145/3152041.3152084

Laberge, G., Shirzad, S., Diehl, P., Kaiser, H., Prudhomme, S., Lemoine, A. S., & others. (2019). Scheduling optimization of parallel linear algebra algorithms using supervised learning. In *2019 ieee/acm workshop on machine learning in high performance computing environments (mlhpc)* (pp. 31–43). IEEE. doi:10.1109/mlhpc49564.2019.00009

Pfander, D., Daiß, G., Marcello, D., Kaiser, H., & Pflüger, D. (2018). Accelerating Octo-Tiger: Stellar Mergers on Intel Knights Landing with HPX. In *Proceedings of the international workshop on opencl* (pp. 1–8).

Schäfer, A., & Fey, D. (2008). LibGeoDecomp: A Grid-Enabled Library for Geometric Decomposition Codes. In *Proceedings of the 15th european pvm/mpi users' group meeting on recent advances in parallel virtual machine and message passing interface* (pp. 285–294). Berlin, Heidelberg: Springer-Verlag. doi:10.1007/978-3-540-87475-1_39

Stumpf, M., Diehl, P., Seshadri, M., Kaiser, H., Heller, T., Howard, D., Biddiscombe, J., et al. (2018). *STEllAR-GROUP/hpxcl: Inital release*. Zenodo. doi:10.5281/zenodo.1409043

Tabbal, A., Anderson, M., Brodowicz, M., Kaiser, H., & Sterling, T. (2011). Preliminary design examination of the ParalleX system from a Software and Hardware Perspective. *ACM SIGMETRICS Performance Evaluation Review*, *38*(4), 81–87. doi:10.1145/1964218.1964232

Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., et al. (2018). A Taxonomy of Task-based Parallel Programming Technologies for High-performance Computing. *The Journal of Supercomputing*, *74*(4), 1422–1434.

Tohid, R., Wagle, B., Shirzad, S., Diehl, P., Serio, A., Kheirkhahan, A., Amini, P., et al. (2018). Asynchronous execution of python code on task-based runtime systems. In *2018 ieee/acm 4th international workshop on extreme scale programming models and middleware (espm2)* (pp. 37–45). IEEE. doi:10.1109/espm2.2018.00009

Wagle, B., Kellar, S., Serio, A., & Kaiser, H. (2018). Methodology for Adaptive Active Message Coalescing in Task Based Runtime Systems. In *2018 ieee international parallel and distributed processing symposium workshops (ipdpsw)* (pp. 1133–1140). doi:10.1109/ipdpsw.2018.00173

Wagle, B., Monil, M. A. H., Huck, K., Malony, A. D., Serio, A., & Kaiser, H. (2019). Runtime adaptive task inlining on asynchronous multitasking runtime systems. In *Proceedings of the 48th international conference on parallel processing* (pp. 1–10). doi:10.1145/3337821.3337915

Zhang, T., Shirzad, S., Diehl, P., Tohid, R., Wei, W., & Kaiser, H. (2019). An introduction to hpxMP: A modern openmp implementation leveraging hpx, an asynchronous many-task system. In *Proceedings of the international workshop on opencl* (pp. 1–10).

Zhang, T., Shirzad, S., Wagle, B., Lemoine, A. S., Diehl, P., & Kaiser, H. (2020). Supporting openmp 5.0 tasks in hpxMP–a study of an openmp implementation within task based runtime systems. *arXiv preprint arXiv:2002.07970*.