Evaluating HPX and Kokkos on RISC-V using an Astrophysics Application Octo-Tiger

Patrick Diehl
Center of Computation & Technology
Louisiana State University
Baton Rouge, Louisiana
USA
pdiehl@cct.lsu.edu

Gregor Daiß
Center of Computation & Technology
Louisiana State University
Baton Rouge, Louisiana
USA
gdaiss1@lsu.edu

Steven R. Brandt
Center of Computation & Technology
Louisiana State University
Baton Rouge, Louisiana
USA
sbrandt@cct.lsu.edu

Alireza Kheirkhahan
Center of Computation & Technology
Louisiana State University
Baton Rouge, Louisiana
USA
aliz@cct.lsu.edu

Hartmut Kaiser
Center of Computation & Technology
Louisiana State University
Baton Rouge, Louisiana
USA
hkaiser@cct.lsu.edu

Christopher Taylor
Tactical Computing Labs
Lindsay, Texas, USA
ctaylor@tactcomplabs.com

John Leidel
Tactical Computing Labs
Lindsay, Texas, USA
jleidel@tactcomplabs.com

ABSTRACT

In recent years, computers based on the RISC-V architecture have raised broad interest in the high-performance computing (HPC) community. As the RISC-V community develops the core instruction set architecture (ISA) along with ISA extensions, the HPC community has been actively ensuring HPC applications and environments [10, 29] are supported. In this context, assessing the performance of asynchronous many-task runtime systems (AMT) is essential. In this paper, we describe our experience with porting of a full 3D adaptive mesh-refinement, multi-scale, multi-model, and multi-physics application, Octo-Tiger, that is based on the HPX AMT, and we explore its performance characteristics on different RISC-V systems. Despite the (limited) capabilities of the RISC-V test systems we used, Octo-Tiger already shows promising results and good scaling. We, however, expect that additional hardware support based on dedicated ISA extensions (such as single-cycle context switches, extended atomic operations, and direct support for HPX's global address space) would allow for even better performance results.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; **Distributed programming languages**.

KEYWORDS

RISC-V, HPX, task-based run time system, asynchronous many-task system, Kokkos

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624230

ACM Reference Format:
Patrick Diehl, Gregor Daiß, Steven R. Brandt, Alireza Kheirkhahan, Hartmut Kaiser, Christopher Taylor, and John Leidel. 2023. Evaluating HPX and Kokkos on RISC-V using an Astrophysics Application Octo-Tiger. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624230

1 INTRODUCTION

RISC-V was introduced to the open community in 2015 as an open standard instruction set architecture (ISA), which is an iteration on established reduced instruction set computer (RISC) principles [38]. RISC-V is based on long academic and practical experience, and an international standards committee with broad industry support maintains its design. Currently, most general-purpose CPUs are based on proprietary ISAs, *e.g.* Intel and AMD use the x86 ISA. Riken's Supercomputer Fugaku recently used A64FX CPU based on the Arm ISA. However, none of these ISAs can be used without permission and payment of royalties. RISC-V, on the contrary, is completely open for use by anyone and royalty-free. For these and other reasons, the European Processor Initiative (EPI), which aims to develop a vendor-independent European CPU for high-performance computing, has identified RISC-V as a target for future investment.

This paper explores the porting and performance of the C++ standard library for parallelism and concurrency (HPX) [21, 22] to a RISC-V single-board computer (SBC). HPX is an asynchronous many-task runtime system (AMT) targeting any-scale from single board computers (like Raspberry Pi [19]) to supercomputers (like ORNL's Summit which is based on IBM®'s Power 9™ CPUS [13], CSCS's Piz Daint using Intel® x86 CPUs [4], or Riken's Supercomputer Fugaku using Fujitsu® A64FX CPUs [15]).

The two RISC-V systems mentioned in this paper use development boards and do not have HPC-grade hardware components. This paper uses a SiFive HiFive Unmatched development board

with four physical cores, a Pine64 Star64 RISC-V board with a four physical core StarFive processor (a licensed SiFive JH7110 design), and two VisionFive2 Open Source RISC-V boards with four physical cores for the distributed runs. At the time of this writing, no HPC-grade RISC-V hardware was available. However, in this paper, we already prepare for the release of such HPC RISC-V hardware by both porting our HPC astrophysics application Octo-Tiger and its toolchain (notably HPX) to RISC-V and evaluating their performance on the currently available hardware. In addition, we will investigate the performance of a simple benchmark on RISC-V with traditional HPC CPUs, like Intel, AMD, and Arm A64FX.

The paper is structured as follows: Section 2 discusses the related work. The software stack is introduced in Section 3. Section 4 describes the effort to port the software stack to RISC-V. Our in-house RISC-V cluster is presented in Section 5. Performance measurements are shown in Section 6. The energy consumption is discussed in Section 7. Section 8 finally concludes the paper.

2 RELATED WORK

The RISC-V architecture has been explored for the following applications: cryptography [31], deep learning [25], and internet of things [28]. Different RISC-V CPUs from various vendors are available, and a comparison of selected CPUs is available here [17]. As of this writing, RISC-V has not yet been widely adopted in scientific computing or high performance computing, but future adoption looks promising.

It sometimes takes time for an ISA to gain traction. For comparison, the ARM64 (Armv8-Series) architecture with 64 Bit was released in 2013, but it took until May 2020 for it to be used in Riken's Supercomputer Fugaku Top 500 machine. The RISC-V ISA was released in 2015, but its adoption in the HPC community has been slower than ARM64's.

This paper will discuss the porting of the C++ standard library for parallelism and concurrency (HPX) to RISC-V. Since our performance section focuses on distributed runs on two single-board computers, we will also focus on the related work on distributed asynchronous many-task runtime systems. Notable mentions are Charm++ [24], Chapel [3], Legion [1], and PaRSEC [2]. For a detailed comparison, we refer to [34]. From the programming paradigm, HPX and Charm++ support similar features. However, HPX's API is based on the specification within the C++ standard, and Charm++ is a library implemented in C++. A comparison between Charm++ and HPX is available here [39].

3 SOFTWARE STACK

In this work, we use two benchmarks. First, a simple HPX benchmark, implementing the Maclaurin series—second, a real-world HPX application: Octo-Tiger.

Octo-Tiger is a distributed astrophysical simulation based on HPX to simulate binary star systems (see Figure 1). It features an adaptive, tree-based data-structure, interleaved solvers, and distributed and GPU capabilities. These features and the simulation's demanding compute requirements make Octo-Tiger an exciting benchmark for HPC work.

To provide context and illustrate the application's complexity that we ported to RISC-V, we briefly introduce Octo-Tiger and its main dependencies (HPX and Kokkos) in this section. We specifically focus on how and why each dependency is used in Octo-Tiger.

3.1 HPX

HPX is an Asynchronous Many-Task Runtime System (AMT), written in modern C++ [21, 23]. HPX makes it possible to write High Performance Computing (HPC) codes such as Octo-Tiger in a task-based fashion using C++ futures and continuations. This yields a user-defined task graph. The parallelism within the graph is automatically used to hide communication latencies with remote nodes or synchronizations of GPU results.

HPX offers various node-level and distributed features to accelerate parallel codes. At the node-level, HPX offers lightweight HPX threads (tasks) which can easily be suspended without blocking any OS threads. HPX allows tasks to be chained together using continuations (`hpx::future::then, hpx::when_all`). In this way, a user can build a directed acyclic task graph (DAG) of tasks directly in their source code. Each of these asynchronous functions can return futures, making it possible to asynchronously wait for GPU work or communication. Furthermore, HPX implements the C++ 20 API related to concurrency and parallelism. For instance, there is a `hpx::mutex` one can use instead of the `std::mutex` equivalent, see: [32, 33]. The advantage to the HPX mutex is that the runtime can switch it out instead of simply blocking, allowing worker threads to continue working.

In Octo-Tiger, we use these node-level features to orchestrate kernel launches, data-transfers, and pre- and post-processing tasks for each time-step. This task-based approach is particulary handy for quickly making parallel work available to the system during the tree-traversals.

As mentioned, HPX also includes various distributed features. It includes an Active Global Address Space (AGAS [36]), allowing HPX components to be distributed across multiple compute nodes. Remote function calls for methods of these components are available (with unified syntax between local and remote function calls). Furthermore, there are various useful communication primitives, such as buffers and channels, for data exchange.

Users can choose between the backend (parcelport) HPX uses for communication. Currently, three variants are available: TCP, MPI and LCI.

In Octo-Tiger, we use these features to distribute data across compute nodes (by having one HPX component per tree-node, which can be placed on any compute node available). Thanks to the unified syntax between local and remote function calls, the implementation of distributed tree traversals is greatly simplified, as we do not need to worry about, for example, if a child tree-node is on the same compute node when traversing the tree via recursion. The HPX futures we get from these asynchronous, potentially remote, function calls, and the futures obtained from data exchanges with other compute nodes neatly integrate with the aforementioned node-level task-graph, yielding a natural overlapping of computation and communication when enough tasks are available.

3.2 Using Kokkos in HPX Applications

HPX orchestrates the launches of the various compute kernels and manages their launch and data dependencies (by allowing the user

to define them in the task-graph with futures), but what about the kernels themselves? Given the heterogeneity of the currently available supercomputers, ranging from CPU-only platforms with AVX or SVE SIMD instructions to GPU supercomputers with Intel, AMD, or NVIDIA GPUs, writing portable compute kernels is crucial.

The Kokkos framework offers a programming model for developing such portable kernels [35]. Kokkos provides various abstractions, like execution and memory spaces for various backends targeting both CPUs and GPUs of different vendors, helping developers achieve this desired (performance) portability. Compute kernels written with Kokkos, using Kokkos Views as data-structures, can be adapted for the desired device by compiling them with the appropriate execution and memory spaces. Thus, by simply using the correct memory spaces at compile-time, the same Kokkos compute kernel implementation can be run on NVIDIA GPUs (using Kokkos' CUDA execution space), on AMD GPUs (using the HIP execution space), and on Intel GPUs (using the SYCL execution space).

Of course, Kokkos kernels also work on CPUs. For example, by using a simple serial execution space (with one CPU core simply executing the kernel), or by using a parallel OpenMP execution space. For better performance on CPUs, SIMD types are available: Operations on those compile down to the appropriate SIMD instructions (for example, AX512) when the Kokkos kernel is built for a CPU execution space. Alternatively, they are mapped to scalar operations on GPUs for compatibility, allowing us to run the same kernel with explicit SIMD vectorization on the CPU [27] and scalar instructions on the GPU.

Kokkos works well with HPX due to two integrations. First, it is possible to get HPX futures for asynchronously launched Kokkos kernels, even on GPUs (which required event polling and integration with the underlying APIs such as CUDA [6] and SYCL [5]).

The second integration is more critical for CPUs. There is a Kokkos HPX execution space that runs a Kokkos kernel directly on the HPX worker threads by internally splitting it into HPX tasks. This avoids conflicting thread pools (as we would encounter when trying to use the OpenMP execution space in HPX applications) and provides users with fine-grained control regarding the number of tasks that are required for each kernel (thus also steering the maximum number of cores desired per kernel which is helpful for concurrent kernels). Full system utilization can be achieved by running one large kernel divided into enough tasks for the available cores or by running enough small kernels concurrently, even if each uses one task (and thus one core). This is an ideal fit for an application such as Octo-Tiger with its adaptive data-structure and interleaved solvers as it contains a mix of computing tasks of varying intensity on each node.

We recently ported Octo-Tiger to Kokkos and now offer a Kokkos implementation for all major compute kernels in its solvers. Notably, both the Serial and the HPX execution space work for us when running the kernels on a CPU as each kernel is concurrently invoked for different parts of the octree, meaning we achieve multicore usage even when using the Serial execution space, with each core working on a separate kernel invocation. Right now, Octo-Tiger can still be compiled without Kokkos and simply uses the old (purely HPX) compute kernel implementations. However, we plan to make Kokkos a mandatory dependency and drop the old, redundant kernel implementations.

3.3 Octo-Tiger

To provide context for this work, we briefly introduce Octo-Tiger, its solvers, and its data structure.

Octo-Tiger is used to simulate and study binary star systems and their eventual outcomes. Stars are modeled in Octo-Tiger as self-gravitating astrophysical fluids. The Octo-Tiger simulation required implementing two interleaved solvers, a hydro solver and a gravity solver. The hydro solver uses finite volumes to compute the inviscid Navier-Stokes equations. The gravity solver uses a grid-based fast multipole method to compute the (Newtonian) gravitational field generated by the fluid in each time step.

The simulations take too much computational power to use a regular grid. Therefore, we employ adaptive mesh refinement (AMR) to maximize the resolution in the area between the stars, where the mass transfer takes place. As previously mentioned, we use an adaptive octree as our data structure. Each node in the octree contains a $8 \times 8 \times 8$ sub-grid for computational efficiency. For more astrophysical details, we refer to [26].

Put into context with HPX and Kokkos, this means each tree-node (and its sub-grid) is a single HPX component, meaning we can call its functions (for example, the compute kernels) on parent or child tree-nodes without having to worry about which compute node they reside in by using HPX's remote function calls. The (Kokkos) compute kernels each operate on one sub-grid (and potentially its ghost layers if required) at a time, meaning that in each solver iteration, we invoke each compute kernel numerous times (usually once per sub-grid). This gives us numerous parallel kernel calls (tasks) to overlap with the communication and hide associated latencies. This in turn helps with the distributed scalability.

The computational aspects of Octo-Tiger and its scalability were previously studied on Piz Daint [4] and ORNL's Summit [13]. Recent CPU-related work includes integrating SVE SIMD types for A64FX SIMD [8] and consequently doing distributed runs on the A64FX machines Stonybrook's Ookami and Riken's Supercomputer Fugaku [5]. On the GPU side, we recently worked on dynamic GPU work aggregation [7] and did the backend work to integrate HPX with SYCL, potentially targeting Intel GPUs in the future [5]. While we do not discuss the physics explored by Octo-Tiger in this paper, we note that Astrophysical work with Octo-Tiger include, for example, studies of R Coronae Borealis stars [30] and bipolytropic stars [20].

4 PORTING THE SOFTWARE STACK TO RISC-V

Most of HPX is implemented using ISO C++. However, small portions of the runtime system are implemented using assembly. One portion of HPX that often requires custom assembly is the runtime system's implementation of user-space threads. The HPX context-switching software implementation can optionally utilize `Boost.Context`[1] support or a native independently provided assembly implementation for a targeted ISA. We use `Boost.Context` in our port of HPX in this paper since HPX depends on Boost already.

HPX provides software and hardware timing support. The software implementation is portable and utilizes standard ISO C++.

[1] https://www.boost.org/doc/libs/1_82_0/libs/context/doc/html/index.html

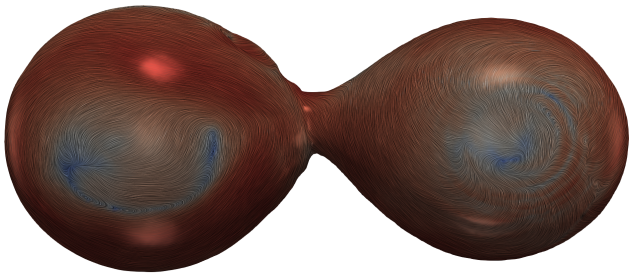


Figure 1: Merger of two stars with the aggregation belt between the two stars. The mass from the donor is transferred to the larger star. The color shows the velocity magnitude of the stream, with red being high velocities. Adapted from [5].

Listing 1: use of rdtime in HPX

```
namespace hpx::util::hardware {

    [[nodiscard]]HPX_HOST_DEVICE inline
    std::uint64_t timestamp()
    {
        std::uint64_t val = 0;
        __asm__ __volatile__(
                "rdtime %0;\n"
                : "=r"(val)
                :: );
        return val;
    }
}    // namespace hpx::util::hardware
```

Hardware-supported implementations require fewer instructions when compared to software implementations and, as a result, experience performance benefits.

The HPX RISC-V port required making a single source code modification in the HPX timer implementation[2]. Listing 1 is the RISC-V-specific portion of HPX that adds hardware-supported timers. The RISC-V HPX port implements timing using the RISC-V `RDTIME` instruction. `RDTIME` is a pseudo-instruction that reads bits from the time Control and Status Register (CSR) [37]. Figure 2 is derived from the RISC-V 2023 draft unprivileged ISA and shows the `RDTIME` instruction format.

The RISC-V port of HPX was initially implemented and tested on a SiFive HiFive Unmatched development board[3] from 2018 using Ubuntu 22.04.2. Table 1 lists the software versions used in this paper. The RISC-V clusters used for this paper include processors made by both the SiFive and StarFive vendors. This paper demonstrates the portability of the HPX RISC-V implementation and the current maturity of RISC-V compiler support.

Porting Kokkos required no changes to the code base, and gcc compiled the library without issues. However, Kokkos's build system CMake files required some minor changes. The RISC-V architecture was not detected, and incorrect compiler flags were added

[2]https://github.com/STEllAR-GROUP/hpx/pull/5968
[3]https://www.sifive.com/boards/hifive-unmatched

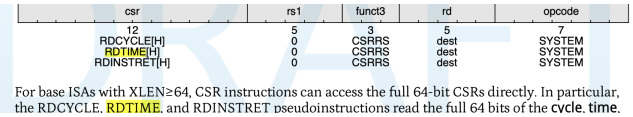


Figure 2: RDTIME instruction from the 2023 Draft Unprivileged ISA. Screenshot from [37].

for the architecture and vectorization. A pull request to add RISC-V support providing the correct compiler flags is in preparation[4]. The RISC-V processors used in this paper do not implement the Vector (V) extension or the SIMD (P) extension in hardware.

As HPX and Kokkos primarily handle the platform-specific details, compiling Octo-Tiger itself for RISC-V was straightforward after these were ported. Our contributions to the HPX and Kokkos toolchains should ease the RISC-V development of other applications.

As a result of this work, HPX, Kokkos, and Octo-Tiger should all be ready to run on future RISC-V-based HPC systems.

5 IN-HOUSE RISC-V TEST SYSTEM

For this paper, we built an in-house cluster using two VisionFive2 Open Source RISC-V[5] Single Board Computer (SBC) computers. Each board has Quad-core StarFive JH7110 64-bit CPU and 8 GB LPDDR4 System Memory. The total cost of this test system was $180. The two boards were connected using the onboard RJ45 Ethernet Connector. Figure 3 shows the small in-house cluster. For better cooling, the cluster was placed in an air-conditioned server room. Starfive provides a snapshot of a Debian image[6] for this board which works perfectly, but this image does not receive any updates. A warning on the documentation explains that updating the image would break the system. Although the official image was working perfectly, the integration was challenging. The Slurm workload manager on the Debian was too old, and the FreeIPA client didn't work correctly. Our attempt to upgrade some of the packages to address the issues broke the system.

Fortunately, Ubuntu Linux provides an alternative operating system for VisionFive2[7]. The Ubuntu image is the 23.04 release compiled for RISC-V boards and receives regular updates and the latest packages like other Ubuntu releases. With this image, FreeIPA client worked perfectly with the FreeIPA on RHEL 8.8, and we could configure the Slurm workload manager. The operating system image was installed on a 64 GB micro SDK with the user home directories and libraries shared over NFS shared file system. As of this writing, the Ubuntu image does not support USB and PCIe on the VisionFive2. The Universal asynchronous receiver-transmitter (UART) serial console was used to configure the network and set up the users, then the rest of the configuration was done over the network. The Star64 and SiFive HiFive Unmatched boards are operated by Tactical Computing Lab and were used to port HPX to RISC-V architecture.

[4]https://github.com/kokkos/kokkos/issues/6323
[5]https://www.starfivetech.com/en/site/boards
[6]https://github.com/starfive-tech/Debian
[7]https://ubuntu.com/download/risc-v

Figure 3: Image of the in-house cluster using two VisionFive2 Open Source RISC-V single board computers with Quad-core StarFive JH7110 64-bit CPU and 8GB LPDDR4 System Memory.

Table 1: Compiler and software versions. The black versions are for the SiFive HiFive Unmatched board and the versions for the Pine64\VisionFive2 boards. On VisionFive2 gcc 12.2.0 **and OpenMPI** 4.1.4 **was used.**

gcc	HPX	Boost	tcmalloc	hwloc
11.3.0	*d1042a9*	1.79/1.82	9.9.5/1.11.13	2.7.0/2.10
Kokkos	HPX-Kokkos	cppuddle	jemalloc	Octo-Tiger
7a18e97	*246b4b8*	*c084385*	5.2.1	*aa38039*

6 PERFORMANCE MEASUREMENTS

In this section, we first compare the performance of HPX's parallel algorithms, sender & receiver, and asynchronous programming on RISC-V on a SiFive HiFive Unmatched from 2018 having a 4-core U74-MC CPU with the performance on Intel, AMD, and Arm CPUs based on previous work. For more details about the implementations, we refer to [14]. Second, we show performance on RISC-V (VisionFive2 Open Source RISC-V) for an astrophysics application, Octo-Tiger, to simulate stellar mergers. Table 1 offers the software and compiler versions.

6.1 HPX Benchmarks

The C++ standard defines approaches for shared memory parallelism: parallel algorithms, asynchronous programming, and senders & receivers. HPX implements these approaches on top of lightweight threads. For all these approaches, we implemented the Maclaurin series for the natural logarithm ln with the basis e, which reads as follows:

$$ln(1+x) = \sum_{n=1}^{\infty}(-1)^{n+1}\frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots, \text{with } |x| < 1 \quad (1)$$

For implementation details, we refer to [14] and to the code on GitHub [16], respectively. Figure **??** shows the measured FLOP\s using asynchronous programming with `hpx::async` and `hpx::future` on all four architectures, namely: Intel Xeon Gold 6140, AMD EPYC 7543, Arm A64FX, and RISC-V SiFive Essential®U74-MC. We look at the node-level scaling, starting with one core on each platform and then increasing the number of cores used for subsequent runs. Given the limited number of cores on the RISC-V boards, this allows for a fairer comparison to the AMD and Intel machines as we equally limit the utilized cores here. This slightly limits the performance gap between the RISC-V single-board computers and these Intel/AMD production-ready CPUs.

The code was executed ten times for each core count, and the reported median time was used to calculate the floating point operations per second. We plotted the variance using error bars with minimal time and maximal time. However, the variance is so small that it is only noticable for a few data points. The basis for the floating point operations was measured to be 100000028581 using a single Intel core using `perf`[8] for $n = 1000000000$. Unfortunately, the RISC-V boards do not yet provide hardware counters to measure more accurate floating point numbers. Therefore, we use the value from the Intel core for all other architectures.

The highest performance was measured on AMD, and the second-highest on Intel. The performance of RISC-V was ≈ 5 times slower than on A64FX but less compared to AMD and Intel. Figure **??** shows the performance using HPX's parallel algorithm `hpx::for_each` with the parallel execution policy `hpx::execution:par`. Here, again, AMD obtained the highest performance, followed by Intel. The performance on RISC-V and A64FX was close but smaller.

For the last approaches, senders & receivers and coroutines, a C++ compiler supporting the C++ 20 standard was needed. Unfortunately, the operating system's default compiler did not support the C++ 20 standard on the Intel and the AMD node. The compilers provided as pre-installed modules on the system did not support the C++ 20 standard either, hence we deferred the tests on those nodes for future work for now. Therefore, Figure 5 only shows results for RISC-V. The sender & receiver implementation performed slightly better than the coroutine implementation. The RISC-V hardware supports Fused Multiply Add (FMA) operations; unfortunately, the RISC-V FMA instructions only support the 32-bit floating point ISA.

To conclude, we have shown that the performance of HPX is around five times less on RISC-V and A64FX. A caveat is that the RISC-V single-board-computer (SBC) is development hardware to port applications to the new CPU architecture. Therefore, the CPU only has four cores. In a previous paper [14] using the logarithm benchmark, we plotted more data. However, we capped the data at ten cores to still show the scaling behavior for the RISC-V boards.

After investigating the FLOP per second, we look into the ratio of floating point operations per second normalized by the theoretical peak performance of each CPU. The theoretical peak performance is given by the clock speed times the vector length times the number of FPU units (# FPU) and the number of cores (# cores)

$$\text{Perf}_{\text{Peak}}(\#\text{cores}) = 2 \times \text{clock speed} \times \text{vector length} \times \#\,\text{FPU} \times \#\,\text{cores}. \quad (2)$$

Most CPUs support fused multiplication and addition (FMA), providing a factor of two in Equation 2. The normalized performance

[8] https://man7.org/linux/man-pages/man1/perf.1.html

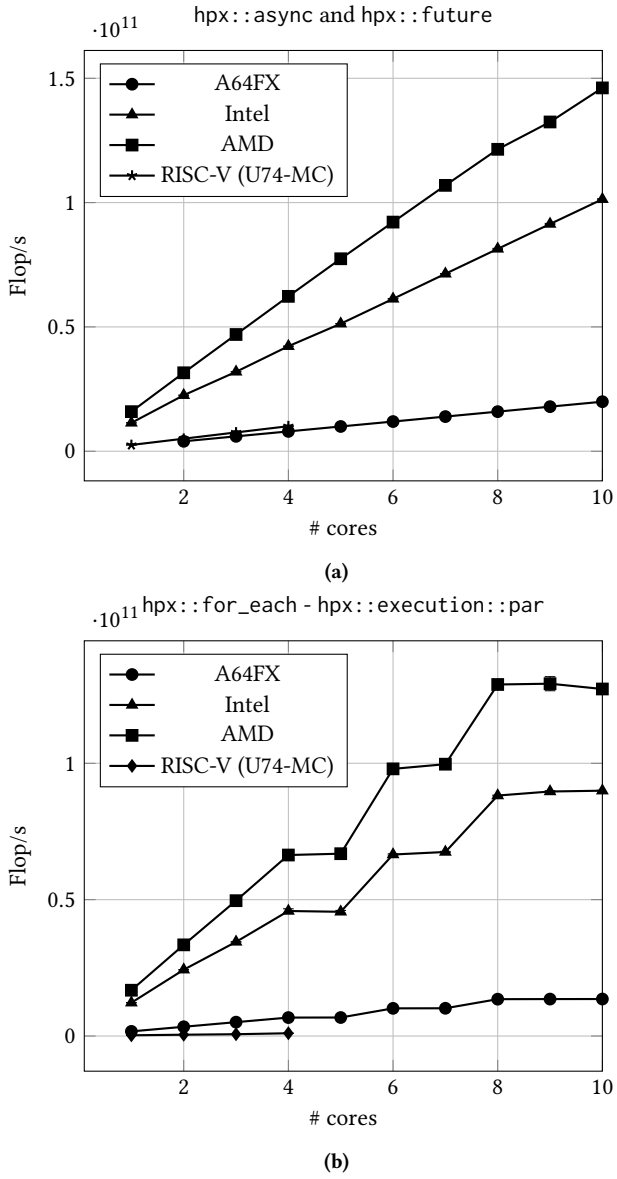


Figure 4: Performance of asynchronous programming using `hpx::async` and `hpx::future` (a) and HPX's parallel `hpx::for_each` (b). The performance measurements for all architectures, except RISC-V, were adapted from [14]. The Taylor series for the natural logarithm in Equation (1) for $n = 1000000000$ and measured 100000028581 floating point operations using *perf* on a single Intel Core.

is given by

$$\text{Perf}_{\text{Norm}}(\#\text{ cores}) = \frac{\text{FLOPs}(\#\text{ cores})}{\text{Perf}_{\text{peak}}(\#\text{cores})}. \quad (3)$$

Table 2 lists the clock speed and the vector length obtained by the vendor's data sheets or via `cat /proc/cpuinfo | grep MHz` for all CPUs used in this study. Figure ?? shows the normalized

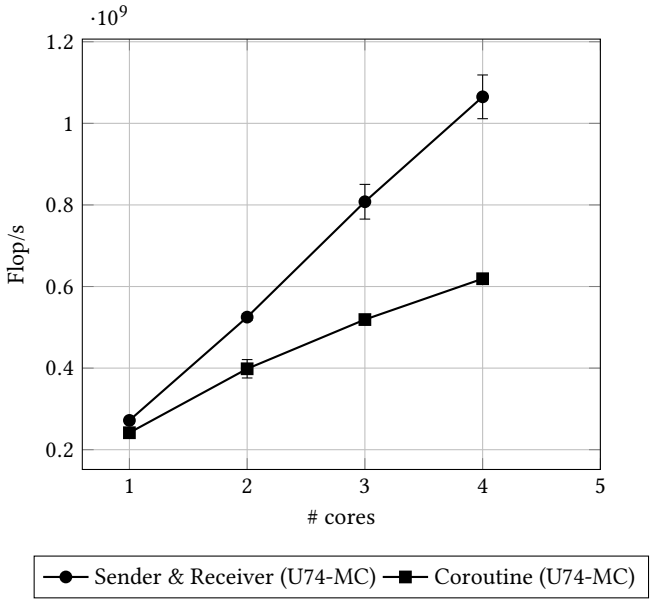


Figure 5: Companions of sender & receiver and future + coroutine parallelism on RISC-V. The Taylor series for the natural logarithm in Equation (1) for $n = 1000000000$ and measured 100000028581 floating point operations using *perf* on a single Intel Core.

performance for asynchronous programming and Figure ?? for parallel algorithms, respectively. For the asynchronous programming and parallel algorithm, we see no effect on auto-vectorization on A64FX. We do not observe a significant effect on Intel and AMD for asynchronous programming. We see some effect on Intel and AMD for the parallel `for` loop. Note that both RISC-V CPU do not support vectorization.

To conclude, the auto-vectorization provided by the GCC compiler does not significantly improve performance. We experienced similar effects for explicit vectorization. For the parallel algorithm, the execution policy `hpx::execution::par_unseq` can be used for implicit vectorization [40]. However, this requires the C++ 20 standard. For A64FX, support for Scalable Vector Extension (SVE) was shown in [9]. From our experience, explicit vectorization resulted in better performance.

6.2 Astrophysics application: Octo-Tiger

After evaluating HPX, we move to the astrophysics application code Octo-Tiger. We ran a single rotating star with gravity and hydro solvers enabled. We start with node-level scaling using the immense amount of cells fitting in one node from one core up to four cores. Next, we show distributed scaling on two SBCs.

6.2.1 Node level scaling. For the node level scaling, a single rotating star with a level of refinement of four is simulated for five-time steps. The octree has 1184 leaf nodes with 606208 cells (as we use 512 cells per sub-grid). Figure 7 shows the scaling from a single core to all four cores. First, Octo-Tiger was compiled using native HPX. This test scaled well. Second, Octo-Tiger was compiled using

Table 2: Clock speed in GHz, vector length, FPU units per core, fused multiplication and addition (FMA) capability, and number of cores for various CPUs obtained by the vendor specification or via `cat /proc/cpuinfo | grep MHz`. Note the FMS is only available for the 32-bit ISA on RISC-V.

CPU	Clock speed [GHz]	Vector length	FPU units per core	FMA	Cores	Peak performance [GFLOP/s]
ARM A64FX	1.8	8	2	✓	48	2764.8
AMD EPYC 7543	2.8	4	2	✓	64	2867.2
Intel Xeon Gold 6140	2.3	8	2	✓	18	1324.8
RISC-V U74-MC(hifiveu)	1.2	NA	1		4	9.6

Kokkos and its HPX Execution Space. This test produced similar scaling. Third, we ran the code using the *Kokkos Serial Execution Space*. This test showed some performance improvement over using the HPX Execution space. This is unsurprising since our concurrent kernel launches give us multicore utilization even when using the Serial execution space. Furthermore, the HPX execution space is mostly beneficial in scenarios where we run enough sub-grids for all CPU cores and hence have to divide compute kernel launches into more HPX tasks for full system utilization.

Compared to HPC-grade nodes, we found that the amount of cells computed per second is small (as is expected given the difference in peak performance). The previous section compared the performance of A64FX with less memory usage. With more memory usage, the slow connection to the memory appears to kick in and slows the overall simulation.

6.2.2 Distributed scaling. After the node level scaling in the previous experiment, we investigated the scaling on our in-house two single-board computer cluster. We used the rotating star level four and executed it on a single board with all four cores and two boards using all four cores on each. As mentioned in Section 3, HPX supports different communication backends. Hence, for our first experiment, we used TCP (Transmission Control Protocol) for communication; and for the second, we used MPI (Message Passing Interface). Figure 8 shows the cells processed per second on a single node and two nodes with different communication backends. The speed-up from a single board to two boards is around 1.85 for TCP and 1.55 for MPI. Both communication backends showed some speed-up relative to a single board. However, the difference between TCP and MPI needs further investigation. In addition, we compared the cells processed per second on the Supercomputer Fugaku, since, in the previous section, the performance for lower memory intense computations was comparable. For a fair comparison, we used only four cores out of the 48 cores of the A64FX CPU. On a single node, we observe that the A64FX CPU using the same amount of cores is ≈ 7 faster.

7 ENERGY CONSUMPTION

We compared the RISC-V boards and Supercomputer Fugaku for the astrophysics application. On Supercomputer Fugaku the power consumption was measured with the PowerAPI [18] interface provided by Riken. On the RISC-V boards, no hardware counters for power measurements are present. Here, we attached a power meter to the USB power source and measured the power consumption while running the Linux command `stress --cpu 4`[9] and while running Octo-Tiger with four cores. As a consequence, the power measurements include power for the entire board. This includes the onboard DRAM memory, SSD storage, Ethernet devices, and other components. Power loss occurs at each step of the transition point, from the wall to the power adapter and from the power adapter to the SBC. The ARM PowerAPI isolates the chip's power consumption measurements, which may explain power measurement differences.

The average power consumption over one minute measured was 3.19 Watt for `stress --cpu 4` and 3.22 Watt for Octo-Tiger. Figure 9 shows the power consumption on both CPU architectures. The power consumption is lower on RISC-V. However, the energy consumption is more significant due to the longer simulation time.

8 CONCLUSION

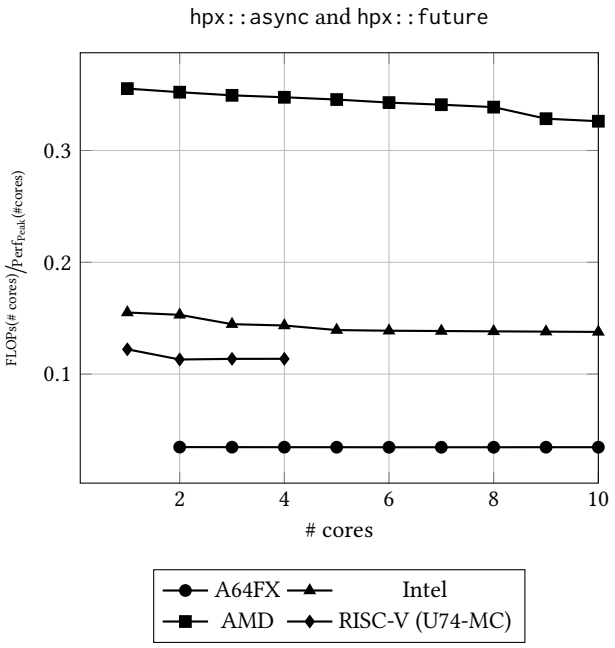
The RISC-V devices used in this paper are single-board computers (SBC) and development board hardware. The RISC-V devices are manufactured with memory controllers and memory channel counts that are not necessarily competitive with the expectations of HPC-grade hardware consumers. HPX scaled on RISC-V SBCs up to four cores. In comparison, HPX scaling did not yield many performance improvements using the third or fourth core from previous experience [19] on Raspberry Pi with an ARM CPU.

Notably, once we got HPX and Kokkos working on RISC-V, we were easily able to obtain a working build of Octo-Tiger, with both the Kokkos compute kernels and its distributed features (via HPX) working. While the currently available hardware limits the performance, this paves the way of using Octo-Tiger as a real-world application benchmark for future (potentially HPC-grade) RISC-V hardware. It allows Octo-Tiger users to consider future RISC-V hardware for production runs/simulations.

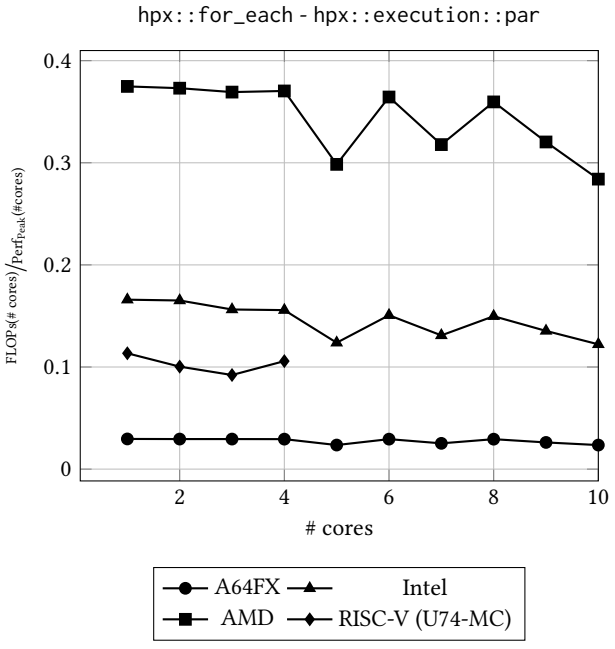
As for the smaller HPX benchmark (implementing the Maclaurin series), the runs with small memory usage on the RISC-V boards were around five times slower than on A64FX (when limited to the same number of cores). However, the RISC-V boards were around seven times slower for more memory-intense simulations. For the distributed runs, a more extensive cluster would be of interest. Due to the cost of $80 per board, this might be expensive for a preliminary portability and evaluation study.

Engineering compromises were made to reduce the single-device price point to move an initial set of devices into the consumer marketplace. The engineering compromises were made to get functional hardware into the marketplace and seed software development efforts such as the development work associated with this paper.

[9] https://linux.die.net/man/1/stress



(a)



(b)

Figure 6: Normalized performance $Perf_{Norm}$(# cores) by the $Perf_{Peak}$(#cores) for asynchronous programming (a) and parallel algorithms (b), respectively.

We expect future iterations of RISC-V hardware will provide competitive performance that matches the expectations of the HPC consumer community. For example, the *Milk-V Pioneer*[10] will have a

[10]https://milkv.io/pioneer

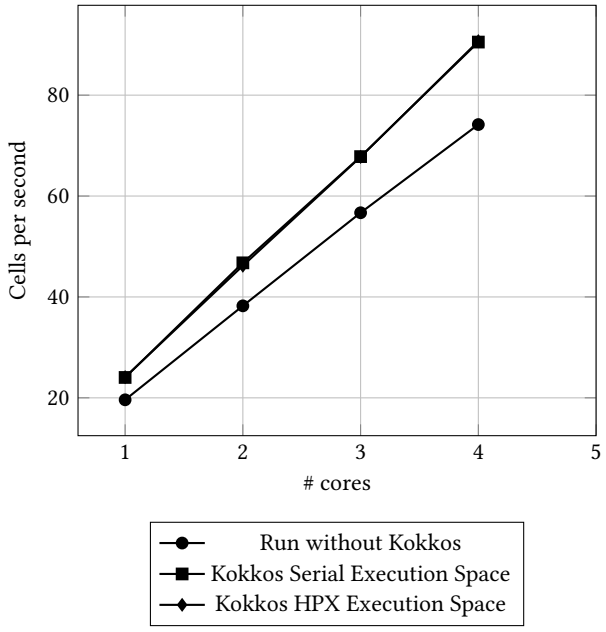


Figure 7: Nodel level scaling from one core to all four cores on a single VisionFive2 for rotating star level four, using Octotiger with three different configurations: Once without Kokkos using the old compute kernels, once using the Kokkos Serial execution space and, lastly, once using the Kokkos HPX execution space

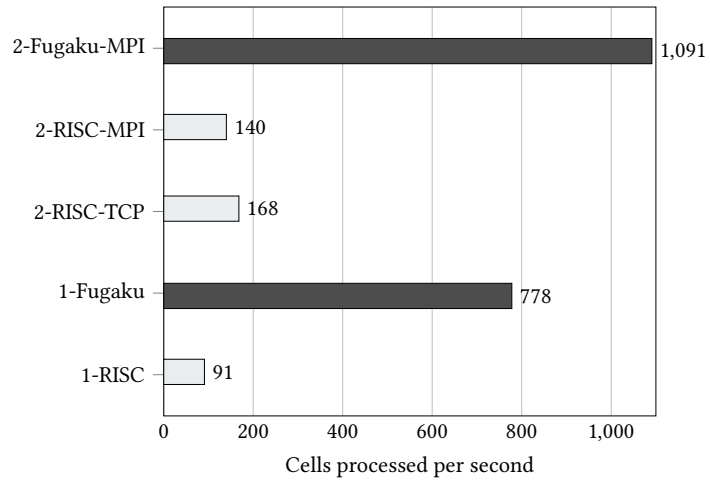


Figure 8: Distributed scaling for a single node and two nodes using TCP or MPI for communication on RISC-V. Unfortunately, our in-house cluster only had two nodes. For a comparison, runs on a single and two Supercomputer Fugaku nodes are shown (each using only four cores out of the 48 available ones for a better comparison).

64-core *SOPHON SG2042* processor advertised as a desktop machine

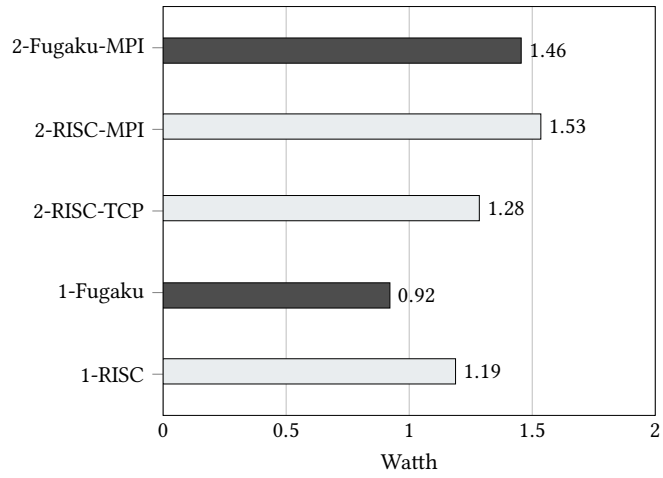


Figure 9: Energy consumption on RISC-V and A64FX CPUs on a single node and two nodes. On Supercomputer Fugaku, the power consumption was measured using PowerAPI. Due to missing hardware counters, the power consumption was measured using a power meter on RISC-V.

for development. However, this machine will have 64 cores for larger scaling runs and improved memory and network controllers. It might not be HPC-grade hardware but it will still be a good improvement.

The benchmark for this study makes heavy use of exponentiation. Exponentiation in RISC-V is performed in software as opposed to hardware. Adding hardware support for exponents can reduce the number of floating point operations from approximately ceil((2*e) + 3) down to 4. We believe micro architectural changes, different pipelining techniques, and additional support for out-of-order execution can significantly impact Flops. Hardware counters on RISC-V to measure floating point operations would be beneficial for more accurate estimates.

This paper demonstrates an opportunity for future work that uses memory system benchmarks (GUPS, STREAM, STREAM-Triad, and LINPACK) to grade the relative performance of RISC-V, development board hardware, and HPC-grade devices.

In addition to the hardware development, the development of ISA extensions is ongoing within the RISC-V community.Some examples that would benefit HPX and other AMTs are one-cycle context switches, extended atomics, hardware support for global address space, and possibly hardware support for thread scheduling (hardware queues).

ACKNOWLEDGMENTS

This work was partly funded by DTIC Contract FA8075-14-D-0002/0007 and the Center of Computation & Technology at Louisiana State University. This research used computational resources of the supercomputer Fugaku provided by the RIKEN Center for Computational Science. 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).

REFERENCES

[1] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 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*. IEEE, IEEE, Salt Lake City, Utah, 1–11.

[2] George Bosilca et al. 2013. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[3] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[4] Gregor Daiß 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* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 62, 37 pages. https://doi.org/10.1145/3295500.3356221

[5] Gregor Daiß, Patrick Diehl, Hartmut Kaiser, and Dirk Pflüger. 2023. Stellar Mergers with HPX-Kokkos and SYCL: Methods of Using an Asynchronous Many-Task Runtime System with SYCL. In *Proceedings of the 2023 International Workshop on OpenCL* (Cambridge, United Kingdom) *(IWOCL '23)*. Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. https://doi.org/10.1145/3585341.3585354

[6] Gregor Daiß, Mikael Simberg, Auriane Reverdell, John Biddiscombe, Theresa Pollinger, Hartmut Kaiser, and Dirk Pflüger. 2021. Beyond fork-join: Integration of performance portable Kokkos kernels with HPX. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, IEEE, virtual event, 377–386.

[7] G. Daiß, P. Diehl, D. Marcello, A. Kheirkhahan, H. Kaiser, and D. Pflüger. 2022. From Task-Based GPU Work Aggregation to Stellar Mergers: Turning Fine-Grained CPU Tasks into Portable GPU Kernels. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 89–99. https://doi.org/10.1109/P3HPC56579.2022.00014

[8] G. Daiß, S. Singanaboina, P. Diehl, H. Kaiser, and D. Pflüger. 2022. From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types. In *2022 IEEE/ACM 7th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE Computer Society, Los Alamitos, CA, USA, 10–19. https://doi.org/10.1109/ESPM256814.2022.00007

[9] Gregor Daiß, Srinivas Yadav Singanaboina, Patrick Diehl, Hartmut Kaiser, and Dirk Pflüger. 2022. From Merging Frameworks to Merging Stars: Experiences using HPX, Kokkos and SIMD Types. In *2022 IEEE/ACM 7th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, Dallas, TX, USA, 10–19. https://doi.org/10.1109/ESPM256814.2022.00007

[10] John Davis. 2021. RISC-V SIG-HPC Enabling RISC-V in HPC, Supercomputers to the Edge, and Emerging AI/ML/DL HPC Workloads. https://riscv.org/blog/2021/06/risc-v-sig-hpc-enabling-risc-v-in-hpc-supercomputers-to-the-edge-and-emerging-ai-ml-dl-hpc-workloads/. Accessed: 2023-08-01.

[11] Patrick Diehl. 2023. Input data: Evaluating HPX and Kokkos on RISC-V using an astrophysics application Octo-Tiger. https://doi.org/10.5281/zenodo.8111772. https://doi.org/10.5281/zenodo.8111772

[12] Patrick Diehl. 2023. *Supplementary materials: Evaluating HPX and Kokkos on RISC-V using an astrophysics application Octo- Tiger*. Ste||ar group. https://doi.org/10.5281/zenodo.8190837

[13] Patrick Diehl et al. 2021. Octo-Tiger's New Hydro Module and Performance Using HPX+ CUDA on ORNL's Summit. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, IEEE, virtual event, 204–214.

[14] Patrick Diehl, Steven R. Brandt, and Hartmut Kaiser. 2023. Shared Memory Parallelism in Modern C++ and HPX. In *Asynchronous Many-Task Systems and Applications*, Patrick Diehl, Peter Thoman, Hartmut Kaiser, and Laxmikant Kale (Eds.). Springer Nature Switzerland, Cham, 27–38.

[15] Patrick Diehl, Gregor Daiß, Kevin Huck, Dominic Marcello, Sagiv Shiber, Hartmut Kaiser, and Dirk Pflüger. 2023. Simulating Stellar Merger using HPX/Kokkos on A64FX on Supercomputer Fugaku. arXiv:2304.11002 [cs.DC]

[16] Patrick Diehl and Chris Taylor. 2023. Shared memory parallelism in Modern C++. https://doi.org/10.5281/zenodo.8067170. https://doi.org/10.5281/zenodo.8067170

[17] Alexander Dörflinger et al. 2021. A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations. In *Proceedings of the 18th ACM International Conference on Computing Frontiers* (Virtual Event, Italy) *(CF '21)*. Association for Computing Machinery, New York, NY, USA, 12–20. https://doi.org/10.1145/3457388.3458657

[18] R. E. Grant, M. Levenhagen, S. L. Olivier, D. DeBonis, K. T. Pedretti, and J. H. Laros III. 2016. Standardizing Power Monitoring and Control at Exascale. *Computer* 49,

10 (Oct 2016), 38–46. https://doi.org/10.1109/MC.2016.308
[19] Nikunj Gupta et al. 2020. Deploying a task-based runtime system on Raspberry Pi clusters. In *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, IEEE, virtual event, 11–20.
[20] Kundan Kadam, Patrick M Motl, Dominic C Marcello, Juhan Frank, and Geoffrey C Clayton. 2018. Numerical simulations of mass transfer in binaries with bipolytropic components. *Monthly Notices of the Royal Astronomical Society* 481, 3 (2018), 3683–3707.
[21] Hartmut Kaiser et al. 2020. HPX-the C++ standard library for parallelism and concurrency. *Journal of Open Source Software* 5, 53 (2020), 2352.
[22] Hartmut Kaiser et al. 2023. *STEllAR-GROUP/hpx: HPX V1.9.0: The C++ Standards Library for Parallelism and Concurrency*. Ste||ar group. https://doi.org/10.5281/zenodo.598202 If you use this software, please cite it using these metadata..
[23] Hartmut Kaiser et al. 2023. *STEllAR-GROUP/hpx: HPX V1.9.1: The C++ Standards Library for Parallelism and Concurrency*. https://doi.org/10.5281/zenodo.8216176 If you use this software, please cite it using these metadata..
[24] Laxmikant V Kale and Sanjeev Krishnan. 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*. ACM, 91–108.
[25] Marcia Sahaya Louis et al. 2019. Towards deep learning using tensorflow lite on risc-v. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, Vol. 1. ACM, Phoenix, Arizona, 6.
[26] Dominic C Marcello, Sagiv Shiber, Orsola De Marco, Juhan Frank, Geoffrey C Clayton, Patrick M Motl, Patrick Diehl, and Hartmut Kaiser. 2021. Octo-Tiger: a new, 3D hydrodynamic code for stellar mergers that uses HPX parallelization. *Monthly Notices of the Royal Astronomical Society* 504, 4 (2021), 5345–5382.
[27] Damodar Sahasrabudhe, Eric T Phipps, Sivasankaran Rajamanickam, and Martin Berzins. 2020. A portable simd primitive using kokkos for heterogeneous architectures. In *Accelerator Programming Using Directives: 6th International Workshop, WACCPD 2019, Denver, CO, USA, November 18, 2019, Revised Selected Papers 6*. Springer, Springer, Denver, Colorado, 140–163.
[28] Pasquale Davide Schiavone et al. 2017. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, IEEE, Thessaloniki, Greece, 1–8.
[29] Agam Shah. 2023. RISC-V Finds Its Foothold in a Rapidly Evolving Processor Ecosystem. https://thenewstack.io/risc-v-finds-its-foothold-in-a-rapidly-evolving-processor-ecosystem/.
[30] Jan E Staff, Brandon Wiggins, Dominic Marcello, Patrick M Motl, Wesley Even, Chris L Fryer, Cody Raskin, Geoffrey C Clayton, and Juhan Frank. 2018. The role of dredge-up in double white dwarf mergers. *The Astrophysical Journal* 862, 1 (2018), 74.
[31] Ko Stoffelen. 2019. Efficient cryptography on the RISC-V architecture. In *Progress in Cryptology–LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2–4, 2019, Proceedings 6*. Springer, Springer, Santiago de Chile, Chile, 323–340.
[32] The C++ Standards Committee. 2017. *ISO International Standard ISO/IEC 14882:2017, Programming Language C++*. Technical Report. Geneva, Switzerland: International Organization for Standardization (ISO). http://www.open-std.org/jtc1/sc22/wg21.
[33] The C++ Standards Committee. 2020. *ISO International Standard ISO/IEC 14882:2020, Programming Language C++*. Technical Report. Geneva, Switzerland: International Organization for Standardization (ISO). http://www.open-std.org/jtc1/sc22/wg21.
[34] Peter Thoman et al. 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* 74, 4 (2018), 1422–1434.
[35] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283
[36] Xi Wang et al. 2021. xBGAS: A Global Address Space Extension on RISC-V for High Performance Computing. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, San Francisco, California, 454–463. https://doi.org/10.1109/IPDPS49936.2021.00054
[37] Asanovic Waterman. 2023. Unprivileged Architecture. In *The RISC-V Instruction Set Manual: Volume I*. University of California, Berkely, CA, USA.
[38] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. 2014. The RISC-V instruction set manual. *Volume I: User-Level ISA', version* 2 (2014), 1–79.
[39] Nanmiao Wu, Ioannis Gonidelis, Simeng Liu, Zane Fink, Nikunj Gupta, Karame Mohammadiporshokooh, Patrick Diehl, Hartmut Kaiser, and Laxmikant V. Kale. 2023. Quantifying Overheads in Charm++ and HPX Using Task Bench. In *Euro-Par 2022: Parallel Processing Workshops*, Jeremy Singer, Yehia Elkhatib, Dora Blanco Heras, Patrick Diehl, Nick Brown, and Aleksandar Ilic (Eds.). Springer Nature Switzerland, Cham, 5–16.
[40] Srinivas Yadav, Nikunj Gupta, Auriane Reverdell, and Hartmut Kaiser. 2021. Parallel SIMD - A Policy Based Solution for Free Speed-Up using C++ Data-Parallel Types. In *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, St. Louis, MO, USA, 20–29. https://doi.org/10.1109/ESPM254806.2021.00008

Listing 2: HPX command line option for the supervisor node using TCP

```
./octotiger --config_file=rotating_star.ini
--max_level=4 --stop_step=5 --theta=0.5
--multipole_host_kernel_type=KOKKOS
--monopole_host_kernel_type=KOKKOS
--hydro_host_kernel_type=KOKKOS
--hpx:agas=10.x.x.160:7910
--hpx:hpx=10.x.x.160:7910
--hpx:localities=2 --hpx:threads=4
```

Listing 3: HPX command line option for the delegate node using TCP

```
octotiger --config_file=rotating_star.ini
--max_level=1 --stop_step=5 --theta=0.5
--multipole_host_kernel_type=KOKKOS
--monopole_host_kernel_type=KOKKOS
--hydro_host_kernel_type=KOKKOS
--hpx:agas=10.x.x.160:7910
--hpx:hpx=10.x.x.168:7910
--hpx:worker --hpx:threads=4
```

A SUPPLEMENTARY MATERIALS

The source code and scripts for benchmarking HPX's feature are available on GitHub[11]. Octo-Tiger[12] and HPX[13] are available on GitHub, respectively. The scripts and input files for the distributed runs are available on GitHub[14] or Zenodo [12], respectively. The input data is available on Zenodo [11].

B DISTRIBUTES RUNS WITHOUT JOB SCHEDULER

The cluster was installed without any job scheduler, like Slurm, and we had to provide the configuration for the distributed runs on the command line. For MPI, we could use the `--hostfile` option and provide the IP addresses of the nodes within the file. For the TCP runs, Listing 2 shows the command line options for the supervisor node, and Listing 3 the command line options for the delegate node, respectively. The blue IP address is the supervisor, and the red is the delegate.

[11] https://github.com/STEllAR-GROUP/parallelnumericalintegration
[12] https://github.com/STEllAR-GROUP/octotiger
[13] https://github.com/STEllAR-GROUP/hpx
[14] https://github.com/diehlpkpapers/RISC-V-23