# Speeding-Up LULESH on HPX: Useful Tricks and Lessons Learned using a Many-Task-Based Approach

Torben Kalkhof, Andreas Koch
*Embedded Systems and Applications Group*
*Technical University of Darmstadt*
Darmstadt, Germany
{kalkhof, koch}@esa.tu-darmstadt.de

***Abstract*—Current programming models face challenges in dealing with modern supercomputers' growing parallelism and heterogeneity. Emerging programming models, like the task-based programming model found in the asynchronous many-task HPX programming framework, offer new ways to express parallelism, enhance scalability, and mask synchronization and communication latency on multi-core and distributed systems.**

**Regular high-performance computing benchmarks are often unsuitable for comparing different programming models due to their limited code complexity. However, real-world scientific applications are usually too complex. As a middle ground, proxy applications model the behavior of actual scientific problems, while reducing code complexity.**

**In our research on using HPX to program machines with heterogeneous compute units (e.g., GPU and FPGA/AI Engines), we have also substantially optimized a pure HPX-based software baseline of the LULESH proxy application. This paper discusses the techniques we applied yielding single-node speed-ups of 1.33x to 2.25x for different problem sizes relative to the LULESH OpenMP reference implementation.**

***Index Terms*—HPC, task-based programming, LULESH, HPX**

## I. Introduction

Modern supercomputers provide enormous computational power, enabling applications in all areas of High-Performance Computing (HPC) that were previously unimaginable. In addition to machine learning and artificial intelligence, these new possibilities greatly benefit applications in all scientific fields. Notable advances include N-body simulation in astrophysics, molecular dynamics simulation, climate modeling, and protein folding.

In recent years, the enhanced computation power of HPC systems is largely due to the continuous growth in parallelism. The total number of compute nodes is increasing, and at the same time, the number of *CPU cores per node* is also rising. Furthermore, there is a growing emphasis on heterogeneity, and new many-core architectures emerge.

The way these systems are programmed has not advanced in the same way. The MPI+X programming model is still the most widely used. However, the underlying idea of dividing and statically scheduling computational problems to different cores and nodes is facing challenges when scaling up to massively parallel systems. The overhead caused by synchronization, communication, and data exchanges between threads and nodes often leads to a waste of computing resources due to waiting execution threads. These issues become even more problematic in the case of load imbalance and irregular applications.

In recent years, there has been a rise in various alternative programming models. One such model is represented by Asynchronous Many-Task runtime systems (AMTs). AMTs involve breaking down a computational problem into countless fine-grained tasks and expressing their dependencies. Unlike a fixed schedule, AMTs dynamically schedule tasks during runtime using the available computational resources. By specifying dependencies at the task level, synchronization barriers can be avoided. Although AMTs cannot completely eliminate communication between nodes, they can effectively reduce latency by overlapping communication and computation. This approach leads to improved overall load balancing, even for irregular applications.

Comparing the performance of different programming models is more challenging than comparing pure hardware performance. Typically, HPC micro-benchmarks such as GEMM or STREAM lack the code complexity needed to properly evaluate new paradigms. On the other hand, porting an entire scientific application to a different programming model requires significant effort and understanding. Proxy applications offer a middle ground. They are simplified versions of real-world applications but maintain the typical structure related to computation and communication. This approach enables the assessment of new programming models using a manageable code base without requiring in-depth knowledge of the underlying scientific context.

In this work, we adapt the LULESH proxy application [1] to use HPX [2], an asynchronous model tasking implemented as a C++ library, as a single-node application. We implement a completely task-based approach and compare our implementation with the OpenMP-based reference implementation [3]. In

This research was funded by the German Federal Ministry for Education and Research (BMBF) with the funding IDs 01 IS 21007 B and 16 KISK 014.

```
1  // create task (executed asynchronously)
2  hpx::future<int> f1 =
3      hpx::async(do_some_work, 42)
4
5  // attach continuation
6  hpx::future<int> f2 =
7      f1.then([](hpx::future<int> &&f_move) {
8          return do_more_work(f_move.get());
9  });
10
11 // create more tasks
12
13 // block until result is ready
14 int result = f2.get();
```

Fig. 1. Minimal code example demonstrating the use of futures and continuations in HPX.

our evaluation using various problem sizes, we achieve speed-ups ranging from 1.33x to 2.25x with our HPX implementation. We show that this speed-up is attained by increasing the overall CPU utilization.

This paper is structured as follows: Section II provides an overview of the HPX programming framework and the LULESH proxy application. Afterward, we discuss related work in Section III. In Section IV, we describe our HPX implementation of LULESH in more detail, before evaluating its performance in Section V. Finally, Section VI concludes our work.

## II. Background

In this section, we first provide an overview of the HPX programming framework, and then describe the LULESH proxy application in more detail.

### A. HPX Programming Framework

HPX [2] is an AMT C++ library. Its API is designed to conform to the C++ standard API for asynchronous execution, while including additional experimental features intended for future standardization.

HPX is based on the concept of *Futurization*. HPX and C++ use *Futures* as objects to hold the state and result of an asynchronous operation. The code example in Figure 1 shows a simple demonstration.

The use of future objects allows for the immediate return when creating an asynchronous task using **`hpx::async()`** (cf. line 2). This enables the current thread to continue executing while the actual computation of the created task will be done later. HPX offers further mechanisms for building task graphs based on futures. *Continuations* make it possible to attach tasks to a future object, and these tasks are executed after the previous task becomes ready (cf. line 6). **`hpx::dataflow()`** or **`hpx::when_all()`** can be used to wait on multiple preceding tasks. In our code example, the functions **`do_some_work()`** and **`do_more_work()`** will be executed asynchronously, and the current execution thread does not block until we finally access the result of the asynchronous operation in line 14.

HPX creates a single *lightweight* thread for each task. These lightweight threads are managed by the HPX runtime entirely in user space, independent of the Operating System (OS).

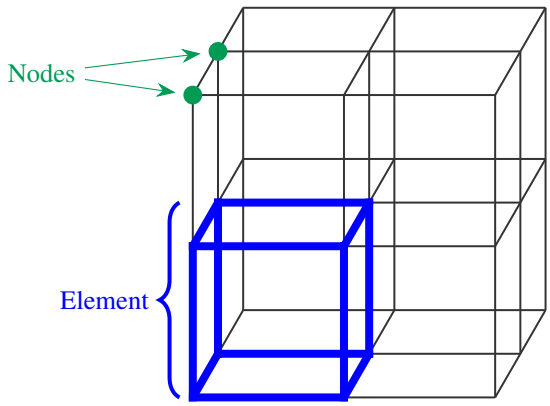


Fig. 2. The hexahedral mesh used in LULESH.

This approach allows for very short context switch times and reduced synchronization overhead compared to traditional "heavyweight" multi-threading at the OS-level. The HPX scheduler runs these many lightweight threads on a smaller set number of OS worker threads, generally one per CPU core. HPX employs techniques like work stealing and sharing to automatically distribute the workload evenly across CPU cores, aiming for high resource utilization.

Furthermore, HPX offers templates for parallel algorithms, such as **`hpx::for_each()`** and **`hpx::reduce()`**. The user provides a custom function, which the runtime can execute in parallel tasks on all input objects based on the specified execution policy.

AMTs target not only local execution, but also *distributed* systems. To simplify task execution on remote nodes, HPX offers a uniform API for remote and local task execution, improving the scalability of parallel programs.

### B. LULESH Proxy Application

The LULESH (Livermore Unstructured Lagrange Explicit Shock Hydrodynamics) proxy application [1] solves the spherical Sedov Blast Wave problem using Lagrange hydrodynamics. Hydrodynamics models the motion of different materials when they are subjected to forces. The computational structure of LULESH is representative of many computer codes that solve hydrodynamics problems. It contains about 4,000 lines of code, making it more complex than the usual HPC micro-benchmarks, but far less complex than most actual scientific codes.

In LULESH, the physical region of interest is represented using a hexahedral mesh as shown in Figure 2. This mesh consists of $s^3$ hexahedrons, called *mesh elements*, and $(s+1)^3$ *nodes* in-between these elements, where $s$ is the size in each dimension. During the calculation, each mesh element can be deformed. Some quantities, such as energy and pressure, are saved for each element, while others such as velocity and position are saved at the node level. As a result, computation steps are performed either node-wise or element-wise.

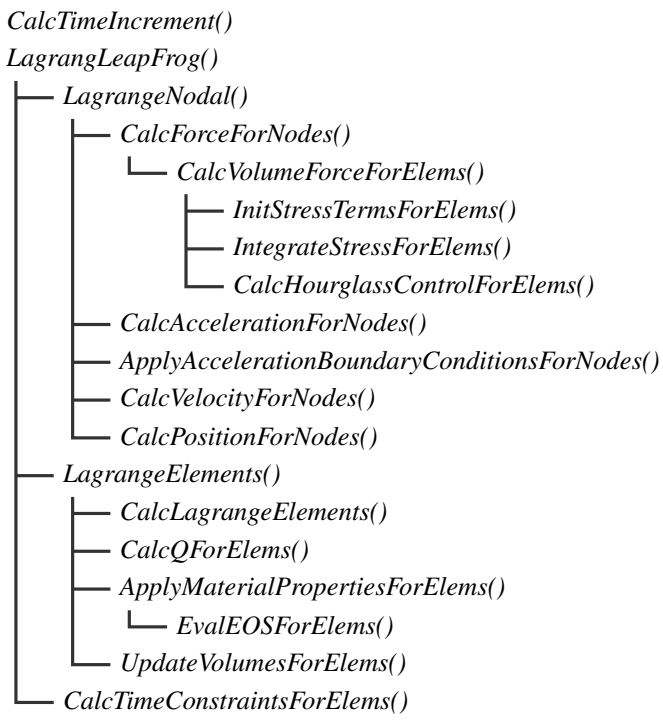


Fig. 3. Simplified call graph of the LULESH proxy application.

In order to model heterogeneous materials, the mesh elements are divided into multiple *regions* with different material properties.

In the following, we provide an overview of the computational structure of LULESH as required to understand this paper. A more detailed description and derivations of the actual calculations are explained in detail in [1].

The main underlying data structure is called *Domain*, which contains arrays for all element and node properties. It also includes a mapping between each mesh element and its surrounding mesh nodes using an array of the respective node indices. Similarly, it has a mapping of all elements that belong to each region.

The computation involves multiple for-loops iterating over the mesh elements or nodes. The OpenMP reference implementation heavily uses `parallel for`-loops. However, some loops are combined into parallel *regions*, resulting in a total of 30 parallel regions.

Figure 3 shows a simplified call graph of LULESH. Each of the listed steps involves multiple kernels. The primary component is the Lagrange *leapfrog* algorithm, which progresses the solution for a single time increment in each iteration. It can be further divided into three steps:

The initial step takes place within the function `LagrangeNodal()`. During this step, the properties of the mesh *nodes* are updated. The most computationally intensive task in this stage involves the calculation of forces at each mesh node in all three dimensions. There are two different force components in LULESH. The first component is induced by stress, and is calculated for each mesh element based on pressure and artificial viscosity. The second force component is based on the Flanagan-Belytschko hourglass control force of a kinematic hourglass filter. Both force components are calculated on an element-by-element basis, and subsequently, the contribution to each surrounding node is determined.

Node acceleration is calculated based on the forces acting on each node. By integrating the acceleration, we can determine the velocity and position of the nodes at this time step.

In `LagrangeElements()`, quantities of the mesh *elements* are updated. First, kinematic element quantities, such as the updated element volume, its derivative, and the deviatoric strain rate tensor, are calculated. Then, the artificial viscosity is updated in `CalcQForElems()`.

In `ApplyMaterialPropertiesForElems()`, the pressure and internal energy of the mesh elements are updated. Unlike other steps, this computation is performed separately for each region rather than on the entire mesh. In real-world scientific applications, the calculation of material properties varies for each material, leading to differences in the computational intensity of the regions. In LULESH, different material properties are modeled by essentially repeating large portions of the `EvalEOSForElems()` subroutine in certain regions, even though the calculation itself remains unchanged. Per default, LULESH doubles the computation for $45\,\%$ of the regions, and increases it even by twenty times for $5\,\%$. Additionally, the speed-of-sound in each mesh element is also determined in this step.

At the end of each iteration, constraints on the maximum time increment for the next iteration are calculated in `CalcTimeConstraintsForElems()`. However, the runtime of this computation step is negligible compared to `LagrangeNodal()` and `LagrangeElements()`.

## III. Related Work

Besides HPX [2], there are several other AMT frameworks. Charm++ [4] is a C++ library that takes a similar approach to HPX. However, its API does not conform to the C++ standard like the HPX API does, but uses custom objects and interfaces instead. StarPU [5] is another AMT library designed for heterogeneous architectures. On the other hand, X10 [6] and Chapel [7] are domain-specific languages to express task-based programs. PaRSEC [8] falls in between, offering a domain-specific language solely for building the task graph.

Legion [9] takes a data-centric approach using so-called data regions. Each task specifies the data region it operates on, and Legion then constructs a data flow and execution graph. Legion can be used directly as C++ library, or with the custom Regent language [10].

The LULESH proxy application [1] is used in various case studies to evaluate programming models, compiler techniques, or hardware architectures. The initial study by Karlin et al. [11] examines and compares multiple emerging programming models, e.g., Charm++ and Chapel, to an OpenMP- and MPI-based implementation. They achieve similar performance using Charm++ compared to the MPI baseline.

The LULESH code is a more challenging application to implement on an AMT, as its structure at first glance leaves little room for load balancing and overlap of communication and computation. We were interested to determine which techniques could be applied to even such "AMT-hostile" code to reap the benefits of high-productivity development using the AMT approach, while still reaching competitive (or ideally: better) performance to the reference codebase.

Further studies include porting LULESH to X10 [12], Chapel once again [13], and LAIK [14]. Ferat et al. [15] use LULESH to investigate task-offloading to GPUs using tasks and the *target* directive in OpenMP.

A prior effort [16] to realize LULESH in HPX primarily just replaced the traditional for-loops with **`hpx::for_each`** constructs. However, this version performs significantly *worse* than the OpenMP reference [17], as confirmed by our own measurements.

In contrast, we employ a completely different approach: We manually divide the computation into tasks and utilize parallelism across loop boundaries in our HPX implementation. This style allows us to *outperform* the OpenMP baseline.

Orthogonal to these efforts of moving LULESH to efficient HPX are initiatives such as [18], which aim to improve the support of HPX for *heterogeneous* computing elements (e.g., GPUs, FPGAs, AI Engines). We do not consider these in the present work, and stay with homogeneous CPU-only execution.

## IV. Implementation

In this section, we describe our task-based approach of implementing the LULESH proxy application using HPX in more detail.

In Section III, we pointed out that simply replacing all OpenMP **`parallel for`**-loop with the corresponding loop constructs provided by HPX can lead to a decrease in performance [17]. In fact, in [16], parallel regions are split into multiple for-loops, which introduces even *more* synchronization barriers. Additionally, LULESH does not expose the existence of load imbalance during its loops, preventing work-stealing between execution threads. Moreover, *creating* HPX threads for every loop implies more overhead than just *distributing* the work on the execution threads in OpenMP. Thus, we need to explore other ways to exploit parallelism and benefit from the asynchronous execution model of HPX.

Figure 4 illustrates the typical structure of an OpenMP program. We use this simplified example of four consecutive kernels throughout this section to demonstrate our changes to the LULESH implementation in a step-by-step fashion. Each **`parallel for`**-loop and parallel region introduces a synchronization barrier between all threads at the end of the loop or region in OpenMP. These synchronization barriers are also present in HPX's parallel for loop constructs.

In our approach, the first step consists of manually breaking down each loop's computation into tasks, instead of just using **`parallel for`**-loop constructs. This concept is illustrated in Figure 5. Each task then iterates over P elements only, with

```
#pragma omp parallel for
for (i = 0; i < N; i++) { ... }

#pragma omp parallel for
for (i = 0; i < N; i++) { ... }

#pragma omp parallel for
for (i = 0; i < N; i++) { ... }

#pragma omp parallel for
for (i = 0; i < N; i++) { ... }
```

Fig. 4. Typical structure of an OpenMP implementation with a sequence of **`parallel for`**-loops.

```
std::vector<hpx::future<void>> v0, v1, v2, v3;
for (i = 0; i < N; i += P) {
    auto f0 = hpx::async([i] (...) {
            for (j = i; j < (i + P); j++) { ... }
    });
    v0.push_back(f0);
}
hpx::wait_all(v0); // synchronization barrier

for (i = 0; i < N; i += P) {
    auto f1 = hpx::async([i] (...) {
            for (j = i; j < (i + P); j++) { ... }
    });
    v1.push_back(f1);
}
hpx::wait_all(v1); // synchronization barrier

for (i = 0; i < N; i += P) {
    auto f2 = hpx::async([i] (...) {
            for (j = i; j < (i + P); j++) { ... }
    });
    v2.push_back(f2);
}
hpx::wait_all(v2); // synchronization barrier

for (i = 0; i < N; i += P) {
    auto f3 = hpx::async([i] (...) {
            for (j = i; j < (i + P); j++) { ... }
    });
    v3.push_back(f3);
}
hpx::wait_all(v3); // synchronization barrier
```

Fig. 5. We manually partition loops into parallel tasks, but maintain the synchronization barrier after each loop.

```
std::vector<hpx::future<void>> v;
for (i = 0; i < N; i += P) {
    auto f1 = hpx::async([i] (...) {
            for (j = i; j < (i + P); j++) { ... }
    }).then([i] (auto &&f) {
            for (j = i; j < (i + P); j++) { ... }
    }).then([i] (auto &&f) {
            for (j = i; j < (i + P); j++) { ... }
    }).then([i] (auto &&f) {
            for (j = i; j < (i + P); j++) { ... }
    });
    v.push_back(f1);
}
// synchronization barrier
auto f_all = hpx::when_all(v);
```

Fig. 6. If no dependencies exist between consecutive loops, we can partition each loop into tasks, and build independent task chains using **`hpx::future<>::then`**.

the partitioning size P being adjustable based on the overall problem size. We use **hpx::async()** to create the tasks and pass the function to be executed as a lambda expression in this example.

In addition, we preserve the synchronization barriers at the end of each loop in this step. They are implemented using **hpx::wait_all()**, which blocks until all futures in the given vector are ready. However, it is important to note that these barriers are often not necessary for the correctness of the calculation itself, but are imposed by the coding structures offered by the programming model.

For example, if we consider the consecutive kernels **CalcVelocityForNodes()** and **CalcPositionForNodes()**, we can observe that both involve element-wise operations. The velocity of each node is calculated based on the acceleration of that node, and similarly the position of each node is only dependent on its velocity. This means that there are no dependencies between *different individual* nodes, and there is no need to delay the calculation of a specific individual node's position until the velocity of *all* other nodes has been calculated.

Thanks to our manual partitioning into tasks, we can immediately launch the tasks calculating the node positions after the task calculating the velocity for this partition of nodes has finished. This structure is shown in Figure 6. In our example, we directly attach the remaining kernels as continuations to the task of the first kernel for each partition. We then need only a single synchronization barrier at the end, after all tasks for all kernels and partitions have been executed.

By creating these independent task chains, we provide the HPX scheduler with more flexibility to improve load-balancing and resource utilization.

We only need seven synchronization barriers in total per iteration. They are required, for example, when a kernel iterating over all mesh *elements* is followed by a kernel iterating over all mesh *nodes*.

Note that we now use **hpx::when_all()** instead of **hpx::wait_all()** to implement the synchronization barrier in our example in Figure 6. Unlike **hpx::wait_all()**, which blocks the execution, **hpx::when_all()** returns a future, becoming ready as soon as all the futures in the passed vector are ready. This enables us to directly attach additional tasks to be executed after the barrier in our LULESH code.

The key computation in LULESH is performed on a central data structure called the *Domain*. However, there are also some temporary variables that require arrays to be allocated in each iteration. This is particularly noticeable during the stress calculation in **LagrangeNodal()**, as well as in the region-wise computation in **ApplyMaterialPropertiesForElems()**. Instead of allocating a single global array for all nodes or elements, we allocate task-local temporary arrays, when possible, to improve data locality.

During implementation, we have observed that some kernels contain relatively little work. Although HPX uses a lightweight threading model, some overhead still exists to create and schedule tasks on the underlying worker OS-threads. Hence,

```
std::vector<hpx::future<void>> v;
for (i = 0; i < N; i += P) {
    auto f1 = hpx::async([i] (...) {
            for (j = i; j < (i + P); j++) { ... }
            for (j = i; j < (i + P); j++) { ... }
    }).then([i] (auto &&f) {
            for (j = i; j < (i + P); j++) { ... }
            for (j = i; j < (i + P); j++) { ... }
    });
    v.push_back(f1);
}
// synchronization barrier
auto f_all = hpx::when_all(v);
```

Fig. 7. Combine consecutive loops in one task to reduce the number of total tasks.

```
std::vector<hpx::future<void>> v;
for (i = 0; i < N; i += P) {
    auto f1 = hpx::async([i] (...) {
            for (j = i; j < (i + P); j++) { ... }
            for (j = i; j < (i + P); j++) { ... }
    });
    v.push_back(f1);
    auto f2 = hpx::async([i] (...) {
            for (j = i; j < (i + P); j++) { ... }
            for (j = i; j < (i + P); j++) { ... }
    });
    v.push_back(f2);
}
// synchronization barrier
auto f_all = hpx::when_all(v);
```

Fig. 8. Tasks for independent consecutive loops can be executed in parallel.

we combine consecutive kernels into one task to reduce the overall number of tasks having to be created. In our example in Figure 7, this is demonstrated by combining two loops into one task each, reducing the number of chained tasks by half. Note that we do *not* fuse the loops of these kernels in order to preserve the computational structure of LULESH, and to thus ensure a fair comparison to the reference implementation. Instead, we maintain multiple separate loops, one after the other, within a single task.

Additionally, we have identified two sections in the *leapfrog* algorithm where independent kernels follow each other. In these sections, the respective tasks can be executed in parallel. The first part of the algorithm involves calculating the forces on the nodes in the mesh. In this step, the two different types of nodal forces, namely volume forces induced by stress and the hourglass forces, can be calculated independently. As a result, we can launch *all* tasks that calculate these forces in parallel. This is illustrated in Figure 8. In this example, instead of attaching the second task as continuation to the first, we create an independent task and add the returned future to our vector of futures. The HPX scheduler is then free to choose the order in which it schedules the tasks. Once calculated, all force components must be summed up for each node.

The second section that allows for parallelization is in **ApplyMaterialPropertiesForElems()**. This part of the algorithm is performed per region. While all kernels for a particular region must be executed sequentially, the different regions are independent of each other and can be calcu-

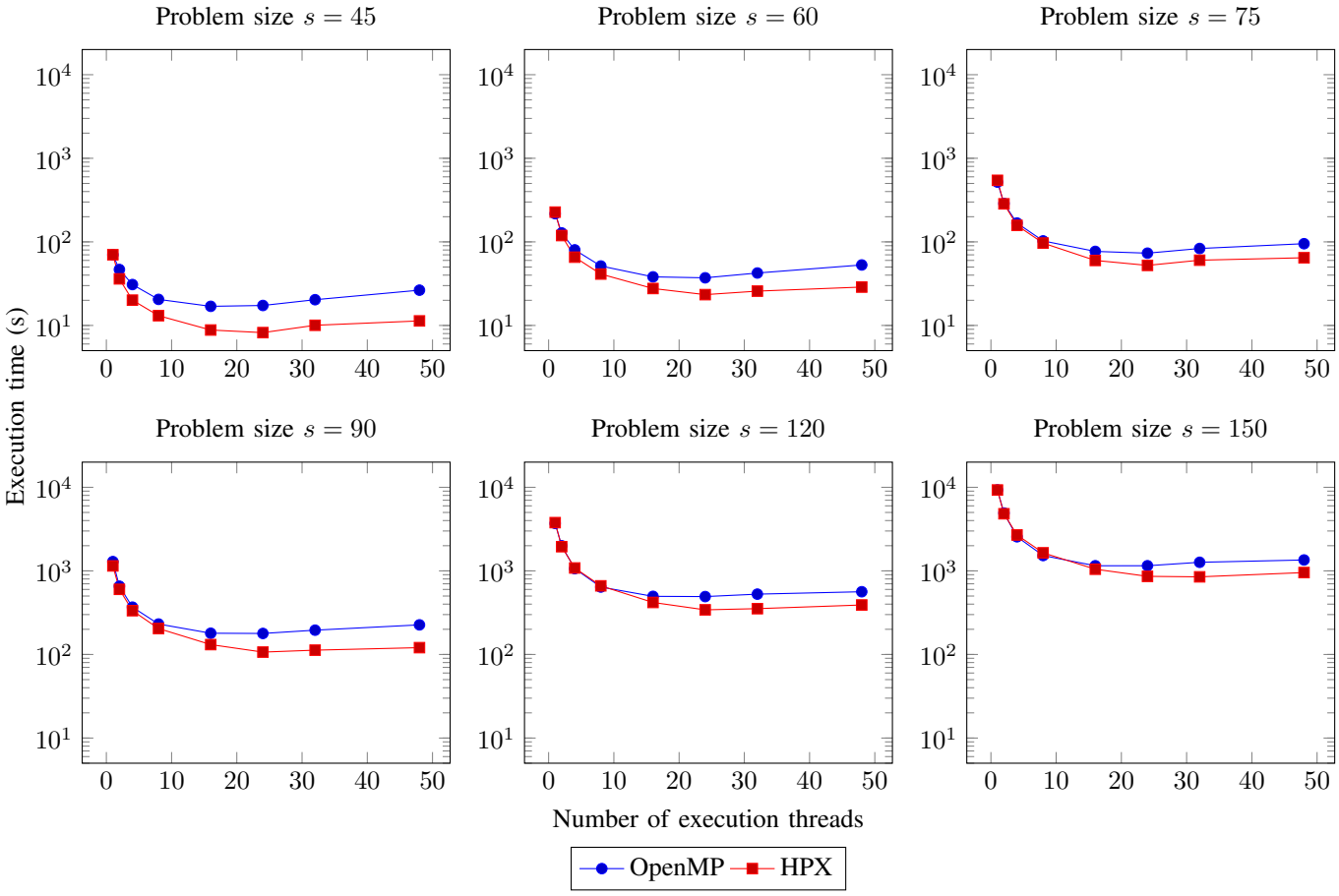


Fig. 9. LULESH runtime of the OpenMP reference and our HPX implementation using different problem sizes and numbers of execution threads. The lowest runtimes are achieved with HPX and 24 threads. Note that the y-axis has a logarithmic scale.

lated in parallel. To this end, we pre-create all tasks for all regions simultaneously and allow the HPX scheduler to balance the work on the worker threads. This works particularly well because there is load imbalance between the regions due to differences in size and computational intensity. As discussed in Section II-B, variances in computational intensity between regions due to different material properties are modeled in LULESH by repeating the calculation of `EvalEOSForElems()` for certain regions.

Overall, our approach of building task chains using *futures* and *continuations* provides us with greater flexibility in expressing parallelism across loops. Additionally, we now pre-create tasks well in advance of the actual execution flow, rather than waiting until the execution flow reaches the respective for-loop. In fact, we pre-create *all* tasks for one iteration of the *leapfrog* algorithm at once, relying heavily on futures and continuations. Basically, the same applies to our short code examples in Figures 6 to 8. These examples only demonstrate how tasks are created. The actual task execution, however, occurs entirely asynchronously outside of these code snippets and is managed by the HPX runtime.

## V. Evaluation

In this section, we compare the performance of our LULESH HPX version to the OpenMP reference implementation. First, we describe our experimental setup and then present our evaluation results. Finally, we report on our programming effort for the realization, to give a (subjective) impression on the productivity of using the AMT paradigm.

All experiments are executed on an AMD EPYC 7443P processor with 24 cores clocked at up to 4035 MHz. Both implementations are compiled using GCC version 13.1.1 with identical optimization flags. Furthermore, we use HPX version 1.10 with JEMalloc [19] as memory allocator. The task scheduling policy being used is HPX's default *priority local scheduling policy*. However, we do not utilize different task priorities. Depending on the problem size $s$, the reported runtimes are the average over 50 runs ($s \leq 75$), 15 runs ($s = 90$), or 2 runs ($s \geq 120$), respectively, due to very long runtimes for larger problem sizes.

### A. Perfomance Results

In our initial experiment, we change both the overall problem size and the number of execution threads. The problem size is determined by the number of mesh elements in each

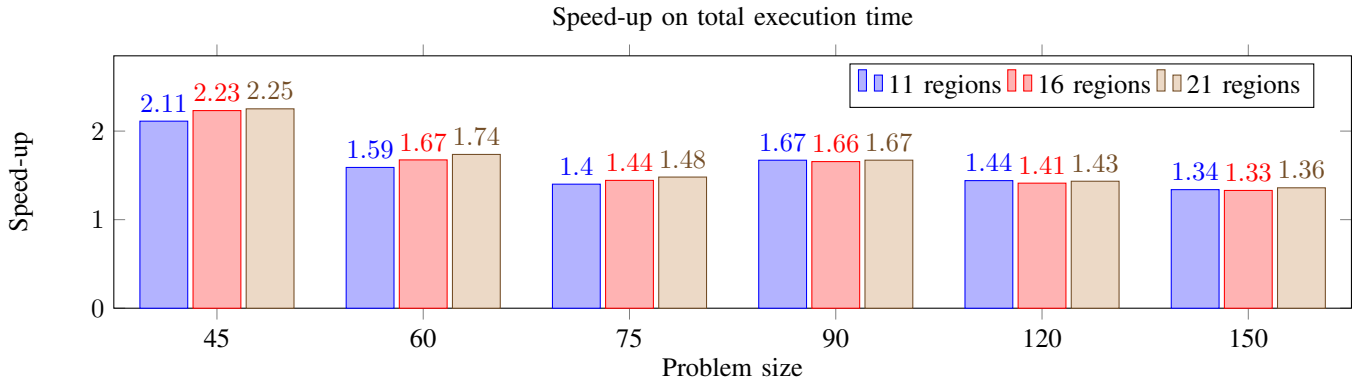


Fig. 10. Speed-up of our HPX implementation in comparison to the OpenMP reference for different problem sizes and number of regions. We use 24 execution threads for both OpenMP and HPX.

dimension. We utilize six different problem sizes: 45, 60, 75, 90, 120, and 150 elements. The number of execution threads is increased in powers of two. In addition, we also test with 24 and 48 execution threads to account for our 24-core CPU. We maintain the default setting of dividing the mesh elements into 11 regions. The actual runtimes of our experiments are shown in Figure 9.

First, we have to examine the behavior of the OpenMP reference implementation. It shows a similar trend when increasing the number of execution threads for all problem sizes. The OpenMP version takes the longest if executed on just a single thread. Runtimes then decrease and reach their minimum at either 16 or 24 threads. With more than 24 execution threads, runtimes slightly increase again. This is most likely due to simultaneous multithreading (SMT) now being required, and the two SMT threads on each CPU core having more interference than speed-up.

In general, our HPX version exhibits similar behavior, consistently achieving the shortest runtimes with 24 threads, meaning we have one worker thread per CPU core and do not incur a penalty due to SMT. The speed-up relative to HPX single-threaded execution is approximately one order of magnitude.

When comparing the OpenMP and HPX versions, we find that the OpenMP version is faster when running with only a single thread. This is because the OpenMP version essentially runs all loops sequentially, while the HPX version incurs an overhead for creating and scheduling many tasks. However, for problem sizes of 45 and 60, we begin to see runtime improvements in our HPX version when using just two execution threads. As we increase the number of OS-threads, the speed-ups also increase, peaking at 24 execution threads (one per core, no SMT).

The OpenMP reference implementation gets only relatively little work done in each loop. For instance, the two consecutive kernels `CalcVelocityForNodes()` and `CalcPositionForNodes()` consist of three multiply-accumulate operations per loop iteration each. When dealing with a small overall problem size, this leads to a lower ratio of productive work vs. synchronization effort. At this point, our implementation benefits from combining consecutive loops into single tasks. Once a task is scheduled, it runs for a longer period until the scheduler needs to be invoked again, resulting in reduced scheduling and synchronization overhead.

When the size of the problem increases further, a larger speed-up is only achieved with 16 or more execution threads. For the two largest problem sizes of 120 and 150, the OpenMP version is even faster with less than 16 threads compared to our HPX version. This is because each loop in the OpenMP version covers a larger range of mesh elements or nodes, resulting in more computation per loop, and a better work-to-synchronization ratio.

Only when utilizing 16 or more execution threads can the HPX scheduler take advantage of the advanced parallelization possibilities in our finely-granular many-tasked implementation. But as the current trend goes towards ever larger per-CPU core counts (e.g., 128 from AMD and Ampere, 288 from Intel), using our HPX "native" AMT approach promises to offer better scalability in the future.

In our second experiment, we analyze the speed-up of our HPX implementation compared to the OpenMP reference implementation. For this comparison, we use a fixed number of 24 execution threads for both implementations. In addition to the first experiment, we also vary the number of regions the mesh elements are divided into. Figure 10 shows the speed-ups of our HPX version for different problem sizes and numbers of regions. In addition to the default of 11 regions, we increment the number by intervals of five and also evaluate with 16 and 21 regions.

The greatest speed-up is observed for the smallest problem size, consisting of 45 mesh elements in each dimension. In this case, our HPX implementation achieves a speed-up of up to 2.25x. We also notice the most significant variation when adjusting the number of regions across all evaluated problem sizes. As previously discussed, our implementation benefits from reduced synchronization overhead as a result of *combining* consecutive loops, while also providing the scheduler with more parallelization flexibility due to finely-

TABLE I
NUMBER OF TASKS IN EACH PARTITION FOR DIFFERENT PROBLEM SIZES USED IN OUR EVALUATION

| Problem size | `LagrangeNodal()` | `LagrangeElements()` |
|---|---|---|
| 45 | 2048 | 2048 |
| 60 | 4096 | 2048 |
| 75 | 8192 | 4096 |
| 90 | 8192 | 4096 |
| 120 | 8192 | 2048 |
| 150 | 8192 | 2048 |

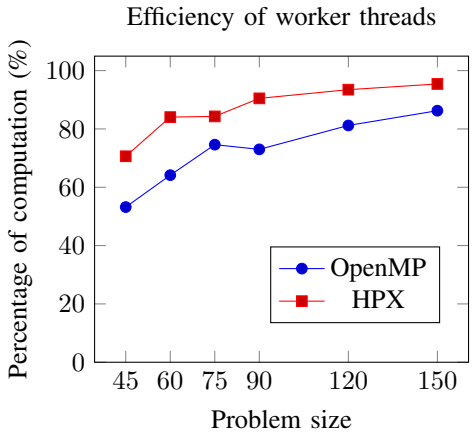


Fig. 11. Average ratio of productive time (during which the worker threads actually perform computations) vs. the total execution time in the OpenMP reference and our HPX implementation.

granular tasks. This effect becomes more pronounced with an increasing number of regions. Since the computation in `ApplyMaterialPropertiesForElems()` is performed separately for each region, the total number of loops increases with more regions, while the number of elements being iterated over in each loop decreases. This leads to even more small loops with a synchronization barrier after each loop. Meanwhile, the number of tasks in our implementation remains similar, as we use a fixed partitioning size for creating the tasks and the total number of mesh elements does not change.

As the problem size increases, the maximum speed-up achieved decreases. The limitations of the OpenMP implementation become less significant. The now-increased workload per loop helps to reduce the synchronization overhead of the OpenMP parallel regions. Additionally, LULESH already is load-balanced, reducing the potential for further performance gains along that axis. However, even with these challenges, we were still able to achieve a speed-up of up to 1.34x for the largest problem size of 150 using our HPX code vs. the OpenMP baseline.

As described in Section IV, our implementation allows us to adjust the partitioning size used for splitting the computation into tasks. The partitioning size significantly impacts performance. If chosen too finely-granular, the total number of tasks increases, leading to rising scheduling overhead and more context switches. In contrast, a too-coarse-grained granularity of tasks may prevent the scheduler from exploiting parallelism and balancing the load between CPU cores efficiently. Through experimentation, we determined that the partitioning sizes listed in Table I are best suited for our application.

We initially increase the partitioning size for the `LagrangeNodal()` part of the *leapfrog* algorithm from 2048, doubling it for each increase in problem size. However, increasing the partition size beyond 8192 does not yield benefits, even though it would reduce the total number of tasks for larger problem sizes. On the other hand, the improved load balancing possibilities compensate for the increased scheduling overhead. Surprisingly, we even experience benefits from *decreasing* the partitioning size in the `LagrangeElements()` part for problem sizes of 120, and 150, to 2048, after using a partitioning size of 4096 for problem sizes of 75 to 90.

In a separate experiment, we compare the resource utilization of both variants. More accurately, we analyze the *ratio* of the productive time, where worker threads actually perform computations, vs. the total execution time, which includes idling, or being engaged in management tasks. In our task-based HPX implementation, we use the respective performance counter provided by HPX to report idle times of the worker threads. In OpenMP, we manually measure the runtime each execution thread spends in each parallel region. With this methodology, we exclude the *single-threaded* portions of the OpenMP implementation from our measurement, but do include the task creation in our HPX implementation. However, we argue that this does not significantly impact our comparison, since all computationally intensive operations are contained in the *parallelized* parts of both implementations. The reported numbers are the average across all execution threads. In contrast to the previous experiments, we do not run LULESH until completion but limit the iterations of the main algorithm due to long execution times, and report the mean over 50 runs.

Figure 11 shows the measured ratio of the time with worker threads performing computations relative to the total execution time. For both variants, we observe an improvement in resource utilization with the increasing problem size. The higher computational effort compensates for the parallelization overhead. In the OpenMP implementation, the worker threads spend only $54\,\%$ of the execution time on computations for the smallest problem size, whereas in our HPX implementation, this percentage is already over $70\,\%$. The HPX implementation then shows slight saturation for problem sizes above 90, with the worker threads spending almost $96\,\%$ on computations. In contrast, the OpenMP implementation does not show a saturation effect. The overhead of statically distributing the computation of parallel regions across execution threads does not increase proportionally to the amount of work, therefore becoming more negligible. Nevertheless, the OpenMP implementation does not achieve a percentage of computation higher than $87\,\%$ in our experiments.

When comparing Figure 10 and Figure 11, we notice a strong correlation between the measured speed-ups and the percentage of computation. Thus, our task-based approach

achieves speed-ups by improving the ratio of actual computations to overhead and idle times of the worker threads.

### B. Programming Effort

When deciding on a programming model, it is important to consider not only performance, but also the size of the programming effort required to use it. Although this is hard to measure, we want to share our subjective experience in porting LULESH to our many-task-based HPX implementation.

Parallelizing a sequential program using OpenMP is quite straightforward. You identify sections, usually loops, that can be executed in parallel, and add the corresponding pragmas in the code. This approach also works using parallel algorithms provided by HPX. However, we have observed that this alone is not enough to achieve comparable or even greater speed-ups when applied 1-to-1 in LULESH.

Instead, we had to take a number of additional steps to be able to exploit more parallelism. Before modifying any code, we needed to conduct a thorough analysis of the fine-grained (per-element/per-node) data dependencies between the different kernels of LULESH. This was necessary in order to identify opportunities for additional parallelization and concurrency beyond loop granularity.

In terms of the code itself, we made minimal changes to the code that carries out the actual calculations. We only made minor adjustments, such as tweaking loop ranges. The primary changes were in the function declarations, as we needed to pass arguments between tasks in different ways, and combine multiple formerly separate functions into one task. However, much effort was put into rewriting the main iteration loop of the leapfrog algorithm. We decided to simplify the call hierarchy, and pre-create all tasks in the top-level function in order to better express dependencies.

Although we would say that the overall code complexity of our HPX implementation is similar to the OpenMP version, the complexity of the code in the main loop that builds the entire task graph for one iteration is much higher than simply calling all kernels sequentially, resulting in about 200 lines of code. This is not only due to the actual code, but also the planning effort involved in the implementation.

Thus, while HPX does enable the expression of more finely-granular parallelism that can be exploited with growing CPU core counts, it takes more intellectual effort on part of the programmer to reason about these structures. Actually implementing in HPX is then relatively easy again, as the AMT framework relies only on a few, well understood primitive constructs.

## VI. Conclusion

In this work, we present a new implementation of the LULESH proxy application using the HPX programming framework. Unlike a previous implementation [16], we have opted for an "HPX native" strictly task-based approach to fully utilize parallelism across loop boundaries. This involves manually dividing loops into tasks and creating task chains using continuations. We then merge consecutive loops into tasks, and execute independent tasks in parallel.

Our evaluation demonstrates that our approach achieves speed-ups compared to the OpenMP reference implementation for all problem sizes. Although LULESH is already well-balanced and quite suitable for parallelization with OpenMP, our approach outperforms OpenMP by improving the overall CPU utilization, with speed-ups of up to 2.25x for the smallest problem size of 45, and around 1.33x speed-up for the largest problem size, even though OpenMP is slightly faster for single-threaded execution.

The decrease in runtime comes with the downside of requiring more programming effort in HPX. We had to to more accurately analyze data dependencies in order to identify separate parts of the program that could run in parallel. Additionally, the process of pre-creating all the necessary tasks and specifying their interdependencies for each iteration of the leapfrog algorithm is significantly more complicated than just calling different kernels of the program in the same sequence as in the original version.

We are convinced that the optimization techniques presented in this work are generally applicable to a wide range of applications. As proxy application, LULESH already represents many scientific codes with a similar structure. However, its general structure of consecutive, parallelizable kernels is not specific just to LULESH-related codes, but is widely used in many kinds of applications. In addition, the remote execution capabilities of HPX ensure scalability not only on single-node but also on multi-node environments.

In future work, our LULESH implementation could be extended to run on multi-node environments and compared to an MPI-based implementation. We anticipate additional benefits from using the asynchronous mechanisms of HPX instead of the mostly synchronous data exchange mechanisms of MPI.

The source code of our implementation is available at [20].

## References


[1] R. D. Hornung, J. A. Keasler, and M. B. Gokhale, "Hydrodynamics Challenge Problem," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-490254.

[2] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, "HPX - The C++ Standard Library for Parallelism and Concurrency," *Journal of Open Source Software*, vol. 5, no. 53, p. 2352, 2020.

[3] Lawrence Livermore National Laboratory, "LULESH on Github." [Online]. Available: https://github.com/LLNL/LULESH/

[4] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based On C++," *ACM SIGPLAN Notices*, vol. 28, no. 10, pp. 91–108, Oct. 1993. [Online]. Available: https://dl.acm.org/doi/10.1145/167962.165874

[5] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, Feb. 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/cpe.1631

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *ACM SIGPLAN Notices*,

vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: https://dl.acm.org/doi/10.1145/1103845.1094852

[7] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1094342007078442

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013. [Online]. Available: https://ieeexplore.ieee.org/document/6654146

[9] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, UT: IEEE, Nov. 2012, pp. 1–11. [Online]. Available: http://ieeexplore.ieee.org/document/6468504/

[10] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for HPC with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/2807591.2807629

[11] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Cambridge, MA, USA: IEEE, May 2013, pp. 919–932. [Online]. Available: http://ieeexplore.ieee.org/document/6569874/

[12] J. Milthorpe, D. Grove, B. Herta, and O. Tardieu, "Exploring the APGAS Programming Model Using the LULESH Proxy Application," IBM Research, Tech. Rep. RC25555.

[13] R. B. Johnson and J. Hollingsworth, "Optimizing Chapel for Single-Node Environments," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Chicago, IL, USA: IEEE, May 2016, pp. 1558–1567. [Online]. Available: http://ieeexplore.ieee.org/document/7530052/

[14] A. Raoofy, D. Yang, J. Weidendorfer, C. Trinitis, and M. Schulz, "Enabling Malleability for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics using LAIK," *PARS-Mitteilungen*, vol. 35, Nr. 1, pp. 109–120, 2020.

[15] M. Ferat, R. Pereira, A. Roussel, P. Carribault, L.-A. Steffenel, and T. Gautier, "Enhancing MPI+OpenMP Task Based Applications for Heterogeneous Architectures with GPU Support," in *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, M. Klemm, B. R. De Supinski, J. Klinkenberg, and B. Neth, Eds. Cham: Springer International Publishing, 2022, vol. 13527, pp. 3–16, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-031-15922-0_1

[16] W. Wei, "lulesh-hpx on Github." [Online]. Available: https://github.com/weilewei/lulesh-hpx

[17] NERSC, "Performance of C++ Parallel Programming Models on Perlmutter using Lulesh." [Online]. Available: https://docs.nersc.gov/performance/case-studies/lulesh-cpp/

[18] T. Kalkhof, C. Heinz, and A. Koch, "Enabling FPGA and AI Engine Tasks in the HPX Programming Framework for Heterogeneous High-Performance Computing," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, 2024.

[19] "jemalloc on Github." [Online]. Available: https://github.com/jemalloc/jemalloc

[20] Embedded Systems and Applications Group, "Source code of task-based HPX LULESH implementation." [Online]. Available: https://doi.org/10.5281/zenodo.13683379

# Appendix: Artifact Description/Artifact Evaluation

## Artifact Description (AD)

### I. Overview of Contributions and Artifacts

#### A. Paper's Main Contributions

$C_1$ We provide a many-task-based implementation in HPX of the LULESH proxy application, achieving speedups between 1.3x and 2.25x compared to the OpenMP reference implementation.

#### B. Computational Artifacts

$A_1$ https://doi.org/10.5281/zenodo.13683379

| Artifact ID | Contributions Supported | Related Paper Elements |
|---|---|---|
| $A_1$ | $C_1$ | Figures 9 - 10 |

### II. Artifact Identification

#### A. Computational Artifact $A_1$

*Relation To Contributions*

This artifact provides the source code of our many-task-based HPX implementation of the LULESH proxy application. It is used to generate the presented runtimes and speedups compared to the OpenMP reference implementation of LULESH. The underlying data structure is generated by the implementation at runtime. The reference implementation is publicly available on Github.

*Expected Results*

Our HPX implementation should be faster than the OpenMP reference implementation of LULESH for the configurations as given in Figures 9 and 10.

*Expected Reproduction Time (in Minutes)*

The expected time for Artifact Setup is about 30 mins.

Using the exact same parameter as presented in the paper, the expected time for Artifact Execution would take several days. However, by limiting the number of iterations executed by the main algorithm of LULESH, the time for Artifact Execution is about 4 hours. This does not reproduce the total runtimes, but it shows the relative speedup of our implementation compared to the reference implementation.

The expected time for Artifact Analysis is about 30 mins.

*Artifact Setup (incl. Inputs)*

*Hardware:* The experiments were conducted on an AMD EPYC 7443 24-core processor. A processor with at least 16 cores may be used to reproduce the results, however, the runs with 48 threads cannot be executed then.

*Software:* The following software packages are required to reproduce the results:

- jemalloc (latest version, 5.3.0), available at:
  - https://github.com/jemalloc/jemalloc
- HPX (version 1.10.0), available at:
  - https://github.com/STEllAR-GROUP/hpx
- LULESH reference implementation (version 2.0), available at:
  - https://github.com/LLNL/LULESH)
- LULESH HPX implementation of this work, available at:
  - https://doi.org/10.5281/zenodo.13683379

*Datasets / Inputs:* No datasets as inputs are required. The underlying data structure is generated by the evaluation software itself and can be configured using program arguments.

*Installation and Deployment:* The artifacts were compiled using GCC version 13.1.1. Older GCC versions may work, but were not tested. In addition, the following build tools are required:

- CMake
- autogen
- automake
- autoconf

Before compiling the LULESH reference implementation, the patch provided by us should be applied. This sets the same compilation flags used for our implementation for a fair comparison, and adds output for easier result analysis.

*Artifact Execution*

The following table gives an overview of the relevant flags of our HPX implementation of LULESH.

| Flag | Description |
|---|---|
| `--s` | Set problem size |
| `--r` | Set number of regions (default: 11) |
| `--i` | Number of iterations |
| `--q` | Suppress verbose output |
| `--hpx:threads` | Number of execution threads |

Use `--q` for all experiments to suppress verbose output. To reproduce the results of the first experiment presented in Figure 9, the problem size and number of execution threads are varied using the `--s` and `--hpx:threads` flags, respectively. For each size out of [45, 60, 75, 90, 120, 150], run with 1, 2, 4, 8, 16, 24, 32 and 48 threads. You may adjust the number of threads if you have a CPU with more or less cores. The `--r` flag can be left out or set to the default of 11.

The second experiment presented in Figure 10 can be reproduced by fixing the number of threads to 24, but varying the number of regions to 11, 16 and 21 using `--r`.

The same measurements must be conducted for the LULESH reference implementation. The following table show the corresponding flags to be used.

| Flag (HPX-based version) | Corresponding flag (OpenMP reference) |
|---|---|
| `--s` | `-s` |
| `--r` | `-r` |
| `--i` | `-i` |
| `--q` | `-q` |
| `--hpx:threads` | `OMP_NUM_THREADS` |

Note that the number of execution threads for OpenMP is not set by a flag but using the environment variable `OMP_NUM_THREADS`.

In our evaluation, we averaged over multiple runs to retrieve our final results. To fit into the time budget of article evaluation, every measurement should be run only once. In addition, use the `--i` flag to limit the number of iterations done per measurement instead of running until completion for problem sizes larger than 60. The following table shows our suggestion for the number of iterations dependent on the problem size.

| Problem size | Number of iterations |
|---|---|
| 75 | 1500 |
| 90 | 770 |
| 120 | 360 |
| 150 | 180 |

*Artifact Analysis (incl. Outputs)*

Both programs print the result of the run in a CSV-compatible format at the end. Reproduce the plots of the first experiment in Figure 9 by plotting runtime over number of threads for a given problem size.

The speed-ups plotted in Figure 10 can be calculated by dividing the runtime of the reference implementation through the runtime of our HPX-based implementation for a given problem size and number of regions.

# Artifact Evaluation (AE)

## A. Computational Artifact $A_1$

*Artifact Setup (incl. Inputs)*

In the following, we describe all required steps to compile the provided artifact and its dependencies. As alternative to manually preparing the artifact, we provide a script which automatically builds the evaluation software, including fetching and building dependencies and the reference implementation.

*1) Option #1: Build with compile script:*

- `cd scripts && bash compile.sh`

*2) Option #2: Build manually step-by-step:*

- We refer to the base directory where our source files are extracted as `$BASE`, return to the base directory before each stop
- Create required install directories and export environment variables:
  - `export Jemalloc_ROOT=$BASE/jem-install`
  - `export HPX_DIR=$BASE/hpx-install`
  - `mkdir $Jemalloc_ROOT $HPX_DIR`
- Clone *JEMalloc* from https://github.com/jemalloc/jemalloc and build using:
  - `git clone https://github.com/jemalloc/jemalloc.git`
  - `cd jemalloc`
  - `./autogen.sh --prefix=$Jemalloc_ROOT`
  - `make -j && make install`
- Clone *HPX* from https://github.com/STEllAR-GROUP/hpx and checkout tag *v1.10.0*:
  - `git clone https://github.com/STEllAR-GROUP/hpx.git`
  - `cd hpx`
  - `git checkout v1.10.0`
- Create build directory and build using *CMake*:
  - `mkdir hpx-build && cd hpx-build`
  - `cmake -DHPX_WITH_MALLOC=jemalloc -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$HPX_DIR -DHPX_WITH_FETCH_ASIO=ON -DHPX_WITH_FETCH_BOOST=ON -DHPX_WITH_FETCH_HWLOC=ON -DHPX_WITH_EXAMPLES=OFF $BASE/hpx`
  - `cmake --build . --target install -j48`
- Create build directory for LULESH HPX implementation and build using *CMake*:
  - `mkdir build && cd build`
  - `cmake -DCMAKE_BUILD_TYPE=Release $BASE`
  - `make -j`
- Clone LULESH reference implementation from https://github.com/LLNL/LULESH and apply patch:
  - `git clone https://github.com/LLNL/LULESH.git`
  - `cd LULESH`

  - `git apply $BASE/patches/ae.patch`
- Create build directory for LULESH rereference implementation and build using *CMake*:
  - `mkdir build-ref && cd build-ref`
  - `cmake -DCMAKE_BUILD_TYPE=Release $BASE/LULESH`
  - `make -j`

*Artifact Execution*

In the following, we describe how to run the experiments. We also provide two scripts to automatically run all experiments: `run-full.sh` for a full evaluation, and `run-reduced.sh` for the limited time budget of the AE process.

*1) Option #1: Run script:*

- `cd scripts`
- `bash run-reduced.sh`

*2) Option #2: Run manually:* Perform all runs manually with the configurations described in the Artifact Description for both our HPX implementation and the OpenMP reference implementation. You can find the executables in the respective build directories. The following example command runs the HPX implementation with problem size 90 and 16 worker threads for 770 iterations of the main loop.

- `build/lulesh-hpx --s 90 --q --i 770 --hpx:threads=16`

The corresponding command for the reference implementation in the OpenMP directory is as follows:

- `OMP_NUM_THREADS=16 build-ref/lulesh2.0 -s 90 -q -i 770`

Create two separate CSV files, one for the HPX and reference implementation each, with the header `size,regions,iterations,threads,runtime,result`. Write the output of each run into the respective CSV file.

*Artifact Analysis (incl. Outputs)*

The provided Python script in `scripts/generate-graphs.py` generates graphs out of the measurement results corresponding to Figures 9 and 10, and print the respective speed-ups of the second experiment. It expects two CSV files, one with the results of all runs using the HPX implementation and another with all results using the reference implementation. If the CSV files have not been generated with the provided run script, they should be passed as arguments to the python script as follows:

- `cd scripts`
- `python3 generate-graphs.py /path/to/hpx.csv /path/to/reference.csv`

The graphs are saved as image files and can be viewed manually. The absolute runtimes should show a similar trend compared to Figure 9, although the absolute values will diverge when running the reduced number of iterations. However, the speed-ups should be comparable to Figure 10.

The graphs can also be generated manually. For the first experiment, plot the runtime over the number of threads in one plot for each problem size. For the second experiment, first calculate the speed-up by dividing the runtime of the reference implementation through the corresponding runtime of our HPX implementation. The resulting speed-ups can then be plotted for each problem size, or directly compared to the given numbers in Figure 10.