Journal of Scientific Computing (2019) 80:878–902
https://doi.org/10.1007/s10915-019-00960-z

Performance Comparison of HPX Versus Traditional Parallelization Strategies for the Discontinuous Galerkin Method

Maximilian Bremer[1] · **Kazbek Kazhyken**[1] · **Hartmut Kaiser**[2] · **Craig Michoski**[1] · **Clint Dawson**[1]

Received: 2 May 2018 / Revised: 23 February 2019 / Accepted: 16 April 2019 / Published online: 2 May 2019

Abstract
As high performance computing moves towards the exascale computing regime, applications are required to expose increasingly fine grain parallelism to efficiently use next generation supercomputers. Intended as a solution to the programming challenges associated with these architectures, High Performance ParalleX (HPX) is a task-based C++ runtime, which emphasizes the use of lightweight threads and algorithm-dependent synchronization to maximize parallelism exposed by the application to the machine. The aim of this work is to explore the performance benefits of an HPX parallelization versus a MPI parallelization for the discontinuous Galerkin finite element method for the two-dimensional shallow water equations. We present strong and weak scaling results comparing the performance of HPX versus a MPI parallelization strategy on Knights Landing architectures. Our results indicate that for average task sizes of 3.6 ms, HPX's runtime overhead is offset by more efficient execution of the application. Furthermore, we demonstrate that running with sufficiently large task granularity, HPX is able to outperform the MPI parallelization by a factor of approximately 1.2 for up to 128 nodes.

Keywords Parallel computing · Discontinuous Galerkin · Shallow water equations · Manycore computing · Task-based parallelism · Knights Landing

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s10915-019-00960-z) contains supplementary material, which is available to authorized users.

✉ Maximilian Bremer
max@oden.utexas.edu

[1] Oden Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX, USA

[2] Center for Computation and Technology, Louisiana State University, Baton Rouge, LA, USA

1 Introduction

The advent of exascale computing presents a disruptive shift in computer architectures. With the end of Moore's law, applications can no longer rely on improving clock frequencies to further computational capabilities. The continued gains in computational performance are rather obtained through increasing core counts, which have been growing at exponential rates over the past decade and are forecasted to continue this trend in the near future with an exascale simulation expected to manage billions of concurrent threads [38]. On the "Stampede" series of NSF flagship supercomputing clusters at the Texas Advanced Computing Center, this has represented an over fourfold increase in the number of cores per node, going from 16 cores per node on Stampede1 to 68 cores per node on Stampede2.

This growth of concurrency makes application performance increasingly sensitive to synchronization mechanisms common in current parallelization strategies such as bulk-synchronous message passing or fork-join parallelism. Task-based parallelism has been noted as an attractive alternative programming model for handling power-constrained design choices including complex memory hierarchies, node heterogeneity, and asynchrony [1]. Fundamentally, task-based parallelism expresses an algorithm as a directed acyclic graph where the vertices are the application's tasks and the edges represent data-dependencies between tasks. By allowing tasks to be scheduled as soon as their dependencies have been satisfied, task-based parallelism naturally gives rise to behaviors such as work stealing and message latency hiding. There has been extensive work on developing task-based programming models, creating a diverse ecosystem of software packages, e.g. Chapel [4,13], Charm++ [36], HPX [29,35], Legion [6], StarPU [2], and OpenMP starting with Version 3.0 [49].

The aim of this paper is to examine the performance of HPX versus a MPI parallelization. HPX is a standards-oriented C++ runtime system, which emphasizes the use of lightweight threads and algorithm dependent synchronization to utilize the concurrency exposed by these new architectures. HPX has been demonstrated to be highly scalable, scaling out to hundreds of thousands of cores for computational astrophysics simulations [28].

The motivating application for this work is the numerical simulation of large-scale coastal ocean physics, in particular, the modeling of hurricane storm surges. One of the leading simulation codes in this area is the Advanced Circulation (ADCIRC) model, developed by a large collaborative team including some of the co-authors [11,17–19,32,33,43,52]. ADCIRC is a Galerkin finite element based model that uses continuous, piecewise linear basis functions defined on unstructured triangular meshes. The model has been parallelized using MPI and has been shown to scale well to a few thousand processors for large-scale problems [51]. While ADCIRC is now an operational model within the National Oceanographic and Atmospheric Administration's Hurricane Surge On-Demand Forecast System (HSOFS), its performance on future computational architectures is dependent on potentially restructuring the algorithms and software used within the model. Furthermore, ADCIRC provides a low-order approximation and does not have special stabilization for advection-dominated flows, thus requiring a substantial amount of mesh resolution. Extending it to higher-order or substantially modifying the algorithms within the current structure of the code is a challenging task.

With this in mind, our group has also been investigating the use of discontinuous Galerkin (DG) methods for the shallow water equations [12,39,41,42,44–47,53,54], focusing on the Runge–Kutta DG method as described in [14]. We have shown that this model can also be applied to hurricane storm surge [16]. DG methods have potential advantages over the standard continuous Galerkin methods used in ADCIRC, including local mass conservation,

ability to dynamically adapt the solution in both spatial resolution and polynomial order (*hp*-adaptivity), and potential for more efficient distributed memory parallelism [40]. While DG methods for the shallow water equations have not yet achieved widespread operational use, recent results have shown that for solving a physically realistic coastal application at comparable accuracies, the DG model outperformed ADCIRC in terms of efficiency by a speed-up of 2.3 and when omitting eddy viscosity, a speed-up of 3.9 [9]. In this paper, we will focus on the DG method applied to the shallow water equations and examine its parallel performance using both HPX and MPI. Based on the knowledge gained, we plan to extend this work to include additional physics necessary for modeling hurricane storm surge; however, in this paper we will focus on a simple test problem that captures the basic algorithm needed for any DG approximation of a shallow water system. One can expect that our results would extend to the application of Runge–Kutta DG methods for general conservation laws.

2 High Performance ParalleX

As computer architectures adapt to meet the design requirements of an exascale system, one of the most disruptive features of the proposed chips is the deluge of concurrency. Applications need to be able to expose increasingly fine grain parallelism to fully utilize these modern architectures. HPX is a C++ runtime system that is designed to take advantage of this concurrency using a task-based approach.

To motivate the design of HPX, we begin with the issues that HPX attempts to address. The Stellar group[1] has coined the term *S.L.O.W.* to describe behaviors of multithreaded applications that hinder performance. The four components of *S.L.O.W.* are:

1. **S**tarvation: cores idling due to insufficient parallelism exposed by the application,
2. **L**atency: delays induced by waiting on dependencies, e.g. waiting on messages which are sent through a cluster's interconnect,
3. **O**verhead: additional work performed for a multithreaded application which is unnecessary in a sequential implementation,
4. **W**aiting for contention resolution: delays associated with the accessing of shared resources between threads.

In addition, HPX provides an elegant programming model based on and extending the C++ concurrency technical specification. Rather than forcing application developers to write efficient multithreaded code, which can lead to difficult to debug race conditions, HPX interfaces with the application at a task-dependency graph based level, taking care of issues such as scheduling. This guards application developers from common multithread-related pitfalls, and in doing so, increases developer productivity.

The HPX runtime system can be categorized into five major components described in the following subsections. A diagram of these components is displayed in Fig. 1.

2.1 Local Control Objects

As mentioned previously, the application interfaces with HPX at the level of the application's task dependency graph. The basic abstraction used in HPX is *futurization*. The completion of any given function returning a type `T` can be represented by a future, e.g.

```
hpx::future<T> my_future = some_expensive_function();
```

[1] http://stellar-group.org/.

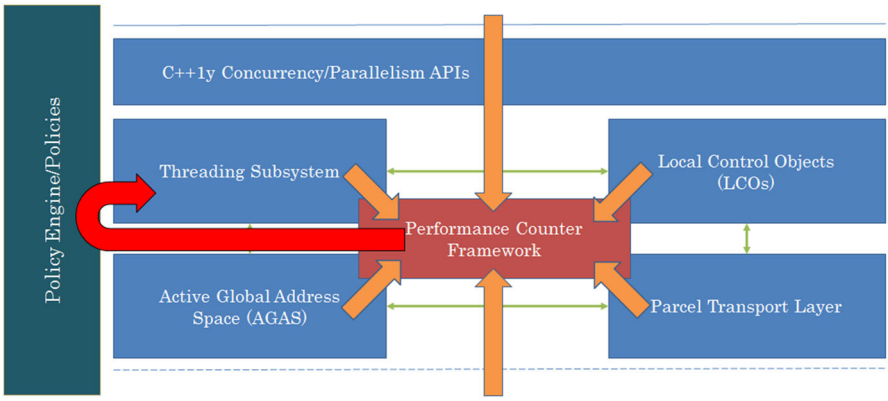


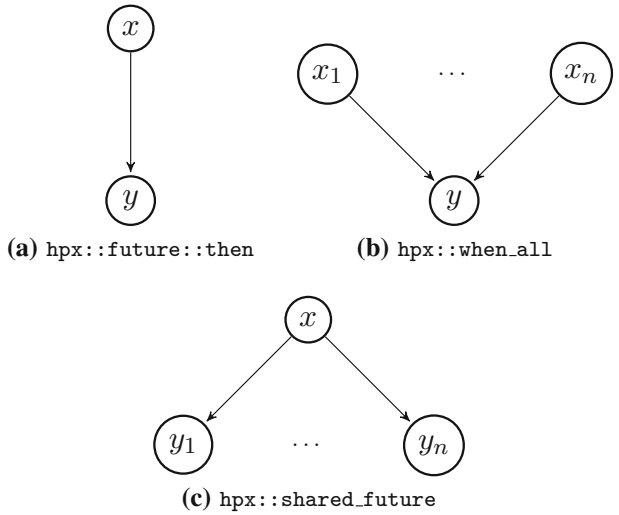
Fig. 1 Software stack diagram of the five major HPX components

When this function is invoked, an HPX-thread is created, which the HPX runtime will schedule and execute. Once the function has been executed, the return value can be retrieved. Upon which, the future is said to have *returned*.

While futures are used to represent the vertices of the application's task-dependency graph, local control objects (LCOs) define the directed edges. One commonly used LCO is the *then* construct. As a member function of `hpx::future`, `hpx::future::then` accepts a function object as its argument, which will be executed upon the returning of the given future. The return type is itself a future, corresponding to the completion of the task being passed in `hpx::future::then`'s argument. The *then* construct allows the result of one function to be used in the evaluation of the continuation without having to explicitly wait for the first future to have returned.

Additional LCOs include `hpx::when_all` and `hpx::when_any`. These LCOs accept a collection of futures as arguments, and return a future, which will be returned either when one or all of the argument futures have returned, respectively. An example use case for `hpx::when_all` would be for a stencil-like kernel. We would like to evaluate the next timestep only after both the internal work and messages have been processed. `hpx::when_all` provides the means to represent this dependency relationship. For a full list of LCOs, we refer the reader to the HPX documentation [35].

Lastly, the task dependency graph can be forked using shared futures or nested parallelism. Since futures are simply C++ objects, the HPX runtime is able to handle nested parallelism simply by instantiating several futures at once. This approach suffices in the case that we begin with one task, and would like to parallelize the evaluation of several sub-tasks nested within the given task. However, in general, we rely on `hpx::shared_future`. One important aspect of `hpx::future` is that once the future has returned, the future's data descopes and becomes inaccessible. In order to support multiple dependents, HPX has introduced the `hpx::shared_future`. For a given shared future, HPX manages the lifetime of the contents of the future in a manner akin to `std::shared_ptr`, descoping the data only after all the dependents have retrieved the shared future's content. Thus, several *then* continuations may be attached to the same shared future, or the shared future may be passed to several LCOs. A pictorial overview of these LCOs is shown in Fig. 2.



(a) `hpx::future::then` **(b)** `hpx::when_all`

(c) `hpx::shared_future`

Fig. 2 Visual representation of how LCOs can be used to generate a task dependency graph in HPX. **a** Shows the input task x and the follow-up task, y. Using the *then* continuation once x has returned, y will be scheduled. **b** Shows $\{x_i\}$ input tasks and one output task, y. Using the `hpx::when_all` construct, when all futures associated with $\{x_i\}$ have returned, y may be executed. **c** Shows the input task x with dependents $\{y_i\}$. Once the shared future x has returned, any y_i may be evaluated. The HPX runtime manages the lifetime of the shared future associated with the output of x, ensuring that the return value of x will remain available for each of the y_i

2.2 Threading Subsystem

Once the task dependency graph has been created, the HPX runtime must execute the tasks, return the futures, and satisfy the dependencies. In doing so, HPX is particularly careful to avoid the pitfalls outlined by *S.L.O.W.*. Given the large overheads associated with spawning and joining operating system threads (OS threads), HPX provides lightweight HPX-threads that execute the tasks associated with the futures created by the application. The scheduler is implemented using an $M : N$ hybrid scheduler [30]. Furthermore, the scheduler has been optimized for rapid context switching of HPX threads. This context switching plays a significant role in mitigating the effects of thread contention and network latencies. If a thread is unable to make progress, the scheduler is able to efficiently switch out the HPX thread and execute a different HPX thread. Ideally, after that thread has finished, the impediment of the original thread's progress will have been resolved, and the application can resume without the core ever idling. This example illustrates one of HPX's design principles: rather than relying on improving interconnect technologies to lower latencies, latencies are hidden by doing useful work until the required dependencies have arrived.

2.3 Active Global Address Space

While the threading subsystem operates within a single private address space, HPX extends its programming model to distributed runs via an *active global address space* (AGAS). Global address spaces attempt to emulate the ease of programming on a single node, while still maintaining tight control over data locality necessary for writing a performant distributed

code. Two well-known global address space models are UPC [22] and Co-Array Fortran [48]. While global address space models like UPC's partitioned global address space (PGAS) are more data-centric, e.g. by exposing pointers to memory addresses on different nodes, HPX approaches global address spaces in a more object-oriented manner.

AGAS consists of a collection of private address spaces, called *localities*. Each locality will run its own instance of the threading subsystem, scheduling threads with locally available resources. In practice, localities are typically chosen to be nodes or non-uniform memory access (NUMA) domains. The basic addressable unit in AGAS is the *component*. Components encapsulate the objects the user would like to remotely access. To interact with a component, the user must go through a smart-pointer-like wrapper class called a *client*. The client can not only manage the component's lifetime via the "Resource acquisition is initialization" (RAII) idiom, but also exposes remotely invokable member functions. Clients can either reside on the same or different locality as their associated component. When a remote locality executes a client member function, HPX will send an active message to the locality where the component is located, execute the function there, and return the result to the client. By interfacing with components through clients, AGAS provides equivalent local and distributed semantics, simplifying the programming of distributed applications.

The key difference between AGAS and other global address space models is its native support for component *migration*. HPX's AGAS layer allows the developer to relocate components to different localities during runtime. With all component functions being invoked through the client interface, HPX is able to ensure that the component functions invoked through the client will be executed on the correct localities, guaranteeing the application's correctness. This functionality can be used to accelerate applications via dynamic load balancing. However, since component migration potentially requires sending large quantities of data through the interconnect, and the load profile is application dependent, HPX requires the application to manage component relocation.

Furthermore, the AGAS does not absolve the programmer of knowledge of where data is located. Since communicating across localities is a relatively expensive operation, the developer is still obliged to minimize this traffic. However, the AGAS guarantees that the task graph will be correctly executed in light of a dynamic distribution of the components across multiple localities.

2.4 Parcel Transport Layer

The parcel transport layer is the abstraction through which HPX sends messages. For this paper, we rely exclusively on the MPI parcelport, which sends messages using `MPI_Isend` and `MPI_Irecv` [30]. However, for optimal interconnect performance, HPX would be able to utilize vendor specific APIs. Note that the implementation of these vendor-specific parcelports through the OpenFabrics Interfaces API [27] are the subject of ongoing work.

2.5 Performance Counter Framework

The last major component of HPX is the performance counter framework. The increase in complexity of both computing architectures and applications has led to challenges in efficiently profiling performance. HPX integrates the process of profiling into the runtime, providing a lightweight mechanism for monitoring application behavior. These counters can not only inspect HPX-related quantities, such as the number of active AGAS components or the length of thread queues, but also provide hooks to hardware counters via PAPI [8]

to directly measure low-level performance features such as memory bandwidth usage and cache misses [26]. With the performance counter framework integrated natively into the HPX runtime system, counters can readily be evaluated by the application, enabling optimizations such as autotuning.

3 Application: The Two-Dimensional Shallow Water Equations

The prediction of hurricane storm surge involves solving physics-based models that determine the effect of wind stresses pushing water onto land and the restorative effects of gravity and bottom friction. These flows typically occur in regimes where the shallow water approximation is valid [21,46]. Taking the hydrostatic and Boussinesq approximations, the governing equations can be written as

$$\begin{aligned} \partial_t \zeta + \nabla \cdot \mathbf{q} &= 0, \\ \partial_t q_x + \nabla \cdot (\mathbf{u} q_x) + \partial_x g(\zeta^2/2 + \zeta b) &= g\zeta \partial_x b + S_1, \\ \partial_t q_y + \nabla \cdot (\mathbf{u} q_y) + \partial_y g(\zeta^2/2 + \zeta b) &= g\zeta \partial_y b + S_2, \end{aligned}$$

where:

- ζ is the water surface height above the mean geoid,
- b is the bathymetry of the sea floor with the convention that downwards from the mean geoid is positive,
- $H = \zeta + b$ is the water column height,
- $\mathbf{u} = [u\ ,\ v]^T$ is the depth-averaged velocity of the water,
- $\mathbf{q} = H\mathbf{u} = \left[q_x\ ,\ q_y\right]^T$ is the velocity integrated over the water column height.

Additionally, g is the acceleration due to gravity, and S_1 and S_2 are source terms that introduce additional forcing associated with relevant physical phenomena, e.g. bottom friction, Coriolis forces, wind stresses, etc.

3.1 The Discontinuous Galerkin Finite Element Method

The discontinuous Galerkin (DG) kernel originally proposed by Reed and Hill [50] has achieved widespread popularity due to its stability and high-order convergence properties. For an overview on the method, we refer the reader to [15,31] and references therein. For brevity, we forgo rigorous derivation of the algorithm, but rather aim to provide the salient features of the algorithm to facilitate discussion of the parallelization strategies.

We can rewrite the shallow water equations in conservation form

$$\partial_t \mathfrak{U} + \nabla \cdot \mathbf{F}(t, \mathbf{x}, \mathfrak{U}) = S(t, \mathbf{x}, \mathfrak{U}), \tag{1}$$

where

$$\mathfrak{U} = \begin{pmatrix} \zeta \\ q_x \\ q_y \end{pmatrix}, \quad \mathbf{F} = \begin{pmatrix} q_x & q_y \\ u^2 H + g(\zeta^2/2 + \zeta b) & uvH \\ uvH & v^2 H + g(\zeta^2/2 + \zeta b) \end{pmatrix}.$$

Let Ω be the domain over which we would like to solve Eq. (1), and consider a mesh discretization $\Omega^h = \cup_e^{n_{el}} \Omega_e^h$ of the domain Ω, where n_{el} denotes the number of elements in the mesh.

We define the discretized solution space, $\mathcal{W}^h$ as the set of functions such that for each state variable the restriction to any element Ω_e^h is a polynomial of degree p. Note that we enforce no continuity between element boundaries. Let $\langle f, g\rangle_\Gamma = \int_\Gamma fg\,\mathrm{d}x$ denote the L^2 inner product over a set Γ. The discontinuous Galerkin formulation then approximates the solution by projecting $\mathfrak{U}$ onto $\mathcal{W}^h$ and enforcing Eq. (1) in the weak sense over $\mathcal{W}^h$, i.e.

$$\langle \partial_t \mathbf{U} + \nabla \cdot F(t, \mathbf{x}, \mathbf{U}) - S(t, \mathbf{x}, \mathbf{U}), \mathbf{w}\rangle_{\Omega^h} = 0$$

for all $\mathbf{w} \in \mathcal{W}^h$, where $\mathbf{U} \in \mathcal{W}^h$ denotes the projected solution. Due to the discontinuities between elements in both trial and test spaces, particular attention must be given to the flux integral, $\langle \nabla \cdot F(t, \mathbf{x}, \mathbf{U}), \mathbf{w}\rangle_{\Omega^h}$, which is not well-defined between elements even in a distributional sense. For evaluation, we define this term as

$$\langle \nabla \cdot F(t, \mathbf{x}, \mathbf{U}), \mathbf{w}\rangle_{\Omega^h} \equiv \sum_{e=1}^{n_{el}} \left(\langle \widehat{\mathbf{F} \cdot \mathbf{n}}, \mathbf{w}\rangle_{\partial\Omega_e} - \langle F(t, \mathbf{x}, \mathbf{U}), \nabla \mathbf{w}\rangle_{\Omega_e} \right),$$

where the boundary integral's integrand is replaced with a numerical flux $\widehat{\mathbf{F} \cdot \mathbf{n}}(\mathbf{U}^{int}, \mathbf{U}^{ext})$ $\mathbf{w}^{int}$. To parse this term, let $\mathbf{U}^{int}$ and $\mathbf{w}^{int}$ denote the value of $\mathbf{U}$ and $\mathbf{w}$ at the boundary taking the limit from the interior of Ω_e^h, and let $\mathbf{U}^{ext}$ denote the value of $\mathbf{U}$ at the boundary by taking the limit from the interior of the neighboring element. For elements along the boundary of the mesh, the boundary conditions are enforced by setting $\mathbf{U}^{ext}$ to the prescribed values. For the numerical flux, $\widehat{\mathbf{F} \cdot \mathbf{n}}$, we use the local Lax–Friedrichs flux,

$$\widehat{\mathbf{F} \cdot \mathbf{n}}(\mathbf{U}^{int}, \mathbf{U}^{ext}) = \frac{1}{2} \left(\mathbf{F}(\mathbf{U}^{int}) + \mathbf{F}(\mathbf{U}^{ext}) + |\Lambda|(\mathbf{U}^{ext} - \mathbf{U}^{int}) \right) \cdot \mathbf{n},$$

where $\mathbf{n}$ is the unit normal pointing from Ω_e^h outward, and $|\Lambda|$ denotes the magnitude of the largest eigenvalue of $\nabla_u \mathbf{F} \cdot \mathbf{n}$ at $\mathbf{U}^{int}$ or $\mathbf{U}^{ext}$.

Since the indicator functions over each element are members of $\mathcal{W}^h$, consider the set B_e for a given element Ω_e,

$$B_e = \left\{ p\mathbf{1}_{\Omega_e} \,:\, p \in \bigoplus_{d=1}^{3} \mathcal{P}^p(\Omega^h) \right\} \subset \mathcal{W}^h,$$

where $\mathcal{P}^p(\Gamma)$ is the set of polynomials of degree p on Γ, and $\mathbf{1}_\Gamma$ is the indicator function over Γ, i.e. $\mathbf{1}_\Gamma(x)$ is 1 if $x \in \Gamma$ and 0 if $x \notin \Gamma$. Since $\{B_e\}_{e=1}^{n_{el}}$ spans $\mathcal{W}^h$, the discontinuous Galerkin method can be alternatively formulated as

$$\partial_t \langle \mathbf{U}, \mathbf{w}\rangle_{\Omega_e^h} = \langle \mathbf{F}, \nabla \mathbf{w}\rangle_{\Omega_e^h} - \langle \widehat{\mathbf{F} \cdot \mathbf{n}}, \mathbf{w}\rangle_{\partial\Omega_e^h} + \langle \mathbf{S}, \mathbf{w}\rangle_{\Omega_e^h} \tag{2}$$

for all $\mathbf{w} \in \bigoplus_{d=1}^{3} \mathcal{P}^p(\Omega_e^h)$ and for all $e = 1, \ldots, n_{el}$.

In order to convey more clearly the implementation of such a kernel in practice, consider the element Ω_e^h. For simplicity of notation for the remainder of the subsection we drop all element-related subscripts, e. Over this element, we can represent our solution using a basis, $\{\varphi_i\}_{i=1}^{n_{dof}}$. Then we can let our solution be represented as

$$\mathbf{U}(t, x) = \sum_{i=1}^{n_{dof}} \tilde{\mathbf{U}}_i(t)\varphi_i(x),$$

where $\tilde{\mathbf{U}}$ are the basis-dependent coefficients describing $\mathbf{U}$. Following the notation of Warburton [23], it is possible to break down Eq. (2) into a set of kernels as

$$\partial_t \tilde{\mathbf{U}}_i = \sum_{j=1}^{n_{dof}} \mathcal{M}_{ij}^{-1} \left(\underbrace{\langle \mathbf{F}, \nabla \varphi_j \rangle_{\Omega_e^h}}_{\mathcal{V}_j} + \underbrace{\langle \mathbf{S}, \varphi_j \rangle_{\Omega_e^h}}_{\mathcal{S}_j} - \underbrace{\langle \widehat{\mathbf{F} \cdot \mathbf{n}}, \varphi_j \rangle_{\partial \Omega_e^h}}_{\mathcal{I}_j} \right), \tag{3}$$

where $\mathcal{M}_{ij} = \langle \varphi_i, \varphi_j \rangle_{\Omega_e^h}$ denotes the local mass matrix. Here we define the following kernels:

- $\mathcal{V}$: The volume kernel,
- $\mathcal{S}$: The source kernel,
- $\mathcal{I}$: The interface kernel.

To discretize in time, we use the strong stability preserving Runge–Kutta methods [24]. Letting

$$\mathcal{L}^h \left(\tilde{\mathbf{U}} \right) = \mathcal{M}^{-1} \left(\mathcal{V} \left(\tilde{\mathbf{U}} \right) + \mathcal{S} \left(\tilde{\mathbf{U}} \right) - \mathcal{I} \left(\tilde{\mathbf{U}} \right) \right),$$

we can define the timestepping method, for computing the i-th stage as

$$\tilde{\mathbf{U}}^{(i)} = \sum_{k=0}^{i-1} \alpha_{ik} \tilde{\mathbf{U}}^{(k)} + \beta_{ik} \Delta t \mathcal{L}^h \left(\tilde{\mathbf{U}}^{(k)} \right),$$

where $\tilde{\mathbf{U}}^{(k)}$ denotes the basis coefficients at the k-th Runge–Kutta stage. We denote the operator, which maps $\left\{ \tilde{\mathbf{U}}^{(k)}, \mathcal{V} \left(\tilde{\mathbf{U}}^{(k)} \right), \mathcal{S} \left(\tilde{\mathbf{U}}^{(k)} \right), \mathcal{I} \left(\tilde{\mathbf{U}}^{(k)} \right) \right\}_{k=0}^{i-1}$ to $\tilde{\mathbf{U}}^{(i)}$ as the update kernel, $\mathcal{U}$.

3.2 Parallelization Strategies

In order to parallelize the DG method, we observe that the evaluation of one Runge–Kutta stage of an element solely depends on the information of the element and its edge-wise neighbors for the previous Runge–Kutta stages. With the intent of parallelization, the DG method can be thought of as a stencil code with an unstructured communication pattern.

We assume that our computing environment can be modeled via a Candidate type architecture. Beyond ensuring that each CPU has access to sufficient work, the candidate type architecture approximation allows us to optimize communication patterns taking into account the difference in intra- and inter-node message latencies. The mesh partitioning is thus broken into 2 phases. An overview of the mesh partitioning strategy is shown in Fig. 3.

In both partitioning phases, we aim to balance the compute load between partitions while minimizing the communication. Since the computational complexity of all kernels is $\mathcal{O}(n_{el})$, the number of elements can be used as a proxy for the computational load. Additionally, we assume that the communication is minimized when the number of edge cuts is minimized. All partitioning is performed using the `METIS_PartGraphKway` function from the METIS library [37].

The first partitioning phase decomposes the mesh into submeshes. For HPX, these submeshes define the granularity of parallelism exposed to the runtime, and ultimately determine the task dependency graph. For MPI, these submeshes correspond to the data assigned to individual MPI ranks. The graph partitioned in this phase uses the mesh's elements as the graph's vertices and places edges between edge-wise connected elements.

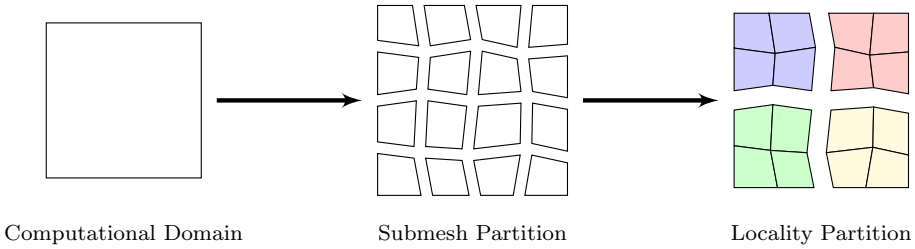


Fig. 3 Overview of the mesh partitioning strategy. The first partitioning phase assigns elements to submeshes, forming our submesh partition. These submeshes are then assigned to localities, balancing the compute load across nodes

The second phase of the graph partitioning assigns submeshes to localities. Since the communication between submeshes is predicated by their shared interface, we can construct a graph from the submeshes of the previous partition. To balance the load, we weight the vertices of this second graph according to the number of elements associated with the relevant submesh, and weight the edges by the number of edge-cuts performed on the element-level graph between two submeshes. For both parallelization strategies, this second partitioning ensures that submeshes which communicate frequently with one another are more likely to be assigned to the same locality.

Beyond balancing the computational load, we include two DG specific optimizations from [3]:

1. Reduction of message sizes, by only sending state variables evaluated at quadrature points along shared interfaces. This reduces message sizes from $\mathcal{O}(p^2)$ to $\mathcal{O}(p)$ per shared interface, where p is the polynomial order of the DG discretization,
2. Hiding message latencies by first sending messages and then computing internal work before waiting for the messages to arrive.

Each Runge–Kutta stage update is broken into two steps as shown in Algorithms 1 and 2. We denote edges whose neighboring elements are assigned to the same submesh as *internal interfaces*, and edges whose neighboring elements are located on different submeshes as *shared interfaces*. Based on the data dependencies, the interface kernel can be evaluated for the internal interfaces as soon as SUBMESH_UPDATE_B of the previous timestep has returned, and thus these evaluations are used to hide send latencies. However, for the shared interfaces, the interface kernel relies on neighboring data, and therefore can only be evaluated once the messages from neighboring submeshes have arrived. These optimizations are applied to both our HPX and MPI implementations. Nevertheless there remain implementation specific details.

3.2.1 HPX Parallelization

For the HPX implementation, each locality maintains a vector of submeshes. The number of elements per submesh determines the *grain size* of parallelism. In selecting the grain size, one must balance two factors. If the grain size is too fine, task overheads will dominate the execution time. On the other hand, if the grain size is too large, we risk exposing insufficient parallelism and performance may suffer due to resource starvation. Due to discrete effects, the grain size is modulated in practice through an *oversubscription factor*, which is defined as the number of submeshes per core on a locality.

Algorithm 1 Compute the first part of the n-th Runge–Kutta Stage update on submesh j

```
function SUBMESH_UPDATE_A(n, j)
    require: SUBMESH_UPDATE_B(n − 1, j) has returned
    Fill all send buffers.
    Post Receives.
    Post Sends.
    for all elements in Submesh j do
        Evaluate S and V
    end for
    for all internal interfaces in Submesh j do
        Evaluate I
    end for
    Wait for all sends and receives to complete. return
end function
```

Algorithm 2 Compute the second part of the n-th Runge–Kutta Stage update on submesh j

```
function SUBMESH_UPDATE_B(n, j)
    require: SUBMESH_UPDATE_A(n, j) has returned.
    for all shared interfaces in Submesh j do
        Evaluate I
    end for
    for all elements in Submesh j do
        Evaluate U
    end forreturn
end function
```

For a given submesh, the evaluation of Algorithms 1 and 2 are futurized, and the futures are chained together via `hpx::future::then` continuations. Furthermore, in Algorithm 1, rather than explicitly waiting for all messages to have been processed, we return a future that will return once all messages have been received and sent.

The use of futurization provides the key advantage of ceding control back to the runtime without unnecessarily suspending the OS thread. While the HPX runtime will internally monitor and process the messages associated with the given submesh, if available, the HPX scheduler will schedule other submesh updates whose dependencies have been satisfied on that OS thread. Thus, even though the locality is waiting for certain messages to arrive before the given submesh update can be completed, the HPX runtime is nonetheless able to use that core to execute tasks associated with other submeshes. Once all the message-related futures have returned, the continuation of the submesh update will be scheduled, and ideally, the application will have progressed without letting cores idle.

3.2.2 MPI Parallelization

The MPI implementation assigns one MPI rank to each core. These ranks communicate with one another via persistent, non-blocking, point-to-point routines. The messages are waited upon using an `MPI_Waitall` with the MPI requests of the local sends and receives as arguments. In the case that the messages have not arrived at the time the `MPI_Waitall` is called, the core will wait on these messages, halting local application progress.

One advantage of this approach is that—similar to HPX—each submesh waits solely on the messages of its neighbors. In doing so, we avoid one of the pitfalls of fork-join parallelism. To ensure application correctness, threads must be synchronized before leaving the fork-join parallel region. With increasing core counts on future architectures, the performance of a

Table 1 Hardware specifications of the Knights Landing architecture on Stampede2

Core	Intel Xeon Phi 7250 ("Knights Landing")
Clock rate	1.4 GHz
Cores per socket	68
Sockets per node	1
Interconnect	Intel Omni-path with fat tree topology
Configuration	Quad-cache

fork-join programming model is limited by Amdahl's law. While using MPI to perform shared memory message passing introduces some unnecessary overhead, the fundamental programming model bypasses this limitation of a fork-join approach.

Lastly, although the send latency is partially hidden by completing internal work before waiting on the messages to arrive, the latency hiding capability of the MPI implementation is limited by the size of the submesh assigned to the MPI rank. With increasing concurrency on future architectures, this MPI parallelization's latency hiding technique will become less efficient.

4 Results

4.1 Experimental Configuration

To assess the advantages of using an HPX over MPI parallelization, we perform strong and weak scaling studies on the Intel Knights Landing (KNL) architecture—a many-core architecture with 68 cores per node. A detailed description of the KNL architecture is provided in Table 1. The software configuration and workflow used to generate the subsequent results are detailed in Appendix A.

As the intent of this paper is not to present high-order methods for hurricane storm surge modeling, but rather a performance comparison of parallelization strategies, we restrict ourselves to solving the 1D inlet problem from [41]. Consider the rectangular domain defined as the Cartesian product $(x_1, x_2) \times (y_1, y_2)$ with $x_1 = 0\,\text{km}$, $x_2 = 90\,\text{km}$, $y_1 = 0\,\text{km}$, $y_2 = 45\,\text{km}$. Additionally, the acceleration due to gravity is set to $g = 9.81\,\text{m/s}^2$, and the bathymetry is constant with depth $H_0 = 3\,\text{m}$. For the boundary conditions, let the superscripts ex/in correspond to the exterior and interior values at the boundary, respectively. Across the boundary $(x_1, y_1) - (x_1, y_2)$, we force a tidal boundary condition, i.e.

$$\zeta^{ex} = A\cos(\Omega t + \eta) \quad \text{and} \quad \mathbf{q}^{ex} = \mathbf{q}^{in},$$

where $A = 0.3\,\text{m}$, $\Omega = 1.405 \cdot 10^{-4}\,\text{rad/s}$, and $\eta = 3\pi/2\,\text{rad}$. At the remaining boundaries, we enforce a land boundary, i.e.

$$\zeta^{ex} = \zeta^{in} \quad \text{and} \quad \mathbf{q}^{ex} \cdot \mathbf{n} = -\mathbf{q}^{in} \cdot \mathbf{n} \quad \text{and} \quad \mathbf{q}^{ex} \cdot \boldsymbol{\tau} = \mathbf{q}^{in} \cdot \boldsymbol{\tau},$$

where $\mathbf{n}$ corresponds to the normal of the boundary, and $\boldsymbol{\tau}$ is the tangent along the boundary.

Each mesh will be a triangulation of the domain $(x_1, x_2) \times (y_1, y_2)$. For each simulation, we generate the mesh by partitioning the domain into an $2N \times N$ Cartesian grid. Each rectangle is then halved along opposite vertices to form a triangular mesh. All simulations are run using a quadratic Dubiner basis [20]. To avoid the CFL condition-related challenges

associated with the various levels of mesh refinement, we select a RKSSP-(2,2) timestepping scheme for the temporal discretization and fix $\Delta t = 0.05$ s with the end-time, $t_{end} = 150$ s.

We exclude initialization from our time measurements, reporting only time spent evaluating Runge–Kutta stages. For the time measurements, we use the `high_resolution_clock` from the STL `chrono` library. The termination of the timings is enforced via a global barrier. For the MPI parallelization, this is achieved by placing an `MPI_Barrier` call after the computation and stopping the timing once MPI rank 0 exits the barrier. For the HPX parallelization, we explicitly wait for all localities to have have finished their computations via an `hpx::wait_all` call, similarly achieving a global synchronization of the application's progress.

In the following sections, we perform strong and weak scaling studies. The execution time for a given run is denoted as ${}^{y}T_{n}^{x}$, where

- y is the type of experiment, with s and w used to indicate strong and weak scaling runs, respectively,
- x is the parallelization strategy: either HPX or MPI,
- n is the number of nodes used for that run.

For the strong scaling study we consider a $1448 \times 724 \times 2$ element mesh and observe the speed-ups obtained by increasing the number of nodes. For a given number of nodes, the mesh is partitioned into two submeshes per core in the case of HPX and one submesh per core for the MPI parallelization. Thus, the communication overhead and task dependency graph grow with the number of cores. In order to compare the performance of the two approaches to one another, we evaluate the parallel efficiency, which we define as

$$E_n^x = \frac{T^*}{Cn^s T_n^x},$$

where ${}^{s}T_{n}^{x}$ is the strong scaling execution time using the parallelization strategy x—HPX or MPI—with n nodes, C is the number of cores per node, and T^* corresponds to the serial execution time. We remark that the serial implementation's performance is strongly affected by the random memory access pattern associated with the unstructured mesh. To mitigate these effects, the serial execution time T^* is obtained by running the HPX parallelization with one thread while still partitioning the mesh into 136 submeshes. This overdecomposition of the mesh effectively acts as a cache blocking mechanism. For reference, the naive serial implementation ran for 122,000 s, and the single-threaded HPX version took 91,000 s. These execution times are based on the average of 10 runs, with the standard deviation being below 0.5% for both cases.

The weak scaling study is done by assigning $1024 \times 512 \times 2$ elements to each node, and then observing the behavior as the number of nodes increases. When the node count does not permit assigning precisely this element count to each node, the nearest number of subdivisions of the domain is chosen to ensure that elements scale linearly with the number of cores, e.g. for two nodes, a mesh with $1448 \times 724 \times 2$ elements was used. For the weak scaling, we use a metric of how many elements are updated by one Runge–Kutta stage per unit time. This is can be thought of as an application specific means of measuring throughput. We begin by analyzing the performance of the partitioning approach outlined in Sect. 3.2. For the remainder of the paper, we refer to this strategy as the *2-phase partitioning strategy*. Thereafter, we present comparison results between HPX and MPI parallelizations.

4.2 Vectorization

One of the key aspects of achieving performance on modern CPU architectures is the ability to effectively utilize SIMD registers. With the KNL's AVX-512 instruction set architecture extension, vectorization potentially allows for a speed-up of $8\times$ for double precision arithmetic. Achieving this speed-up however is hampered by how quickly data can be provided to the processor, and how well the compiler vectorizes the code. Vectorization in the context of DG methods for shallow water flows has been extensively studied in [10]. The method presented therein relies on transforming the code via loop inversion in a way that the compiler is able to generate the optimized binaries. This approach however suffers from maintainability issues that have prevented adoption in the main branch of the code base. For the results presented here, we have utilized the Blaze linear algebra library [34]. This library provides vector and matrix abstractions and combines them with expression templates for efficient evaluation without generating temporary variables. Internally, BLAZE either implements vectorized versions of these basic linear algebra operations, or offloads the calls to a BLAS implementation, e.g. Intel's Math Kernel Library (MKL). Although the vectorization achieved by this approach is nearly identical to that in [10], it is the authors' opinion that code generated via this approach is more maintainable and readable.

Lastly, we remark that for the chosen test cases, the kernel remains memory bandwidth bound. In particular, gathering elements' Gauss point values at the boundaries and scattering the flux values back to the elements during the interface kernel, $\mathcal{I}$, generates random access patterns. Reordering local element indices to minimize element-interface connectivity matrix bandwidth would alleviate some of these issues. However, this remains an issue that we will address in a later work.

4.3 Partitioner Performance

To begin, we benchmark the 2-phase partitioning strategy against a standard flat partitioning strategy. The flat partitioning strategy consists of using METIS to partition the mesh into submeshes, and then assigning submeshes to localities in a round robin manner.

The strong scaling results for the two partitioning approaches are shown in Fig. 4, and the weak scaling results are shown in Fig. 5. Each configuration for the strong and weak scaling studies has been run ten times. For the strong scaling studies, the execution times varied considerably, and have thus been depicted via standard box-and-whisker plots with the whiskers extending up to 1.5 times the inter-quartile range from the relevant quartile.

For the MPI parallelization, the strong scaling results are shown in Fig. 4a and the weak scaling results are shown in Fig. 5a. These scaling results show the flat partitioning approach outperforming the 2-phase partitioning approach by approximately a factor of two. This discrepancy in performance is due the fact that METIS does not strictly satisfy partitioning constraints. In the second phase of the 2-phase partitioning, the number of submeshes assigned to each locality is not strictly equal to the number of cores. Thus, there exist MPI ranks which are assigned two submeshes, while other ranks are assigned none. Since there is no mechanism for work stealing between MPI ranks, the 2-phase approach doubles the length of the critical path of the task executions, causing the observed slow down. This slowdown for the 2-phase partitioning strategy is most clearly observed at 128 nodes for the weak scaling study where ${}^{w}T_{1}^{MPI}/{}^{w}T_{128}^{MPI}$ is 0.50. This aspect of METIS has been extensively studied in [5]. Nevertheless, it is worth noting that the flat partitioning approach proves to be highly

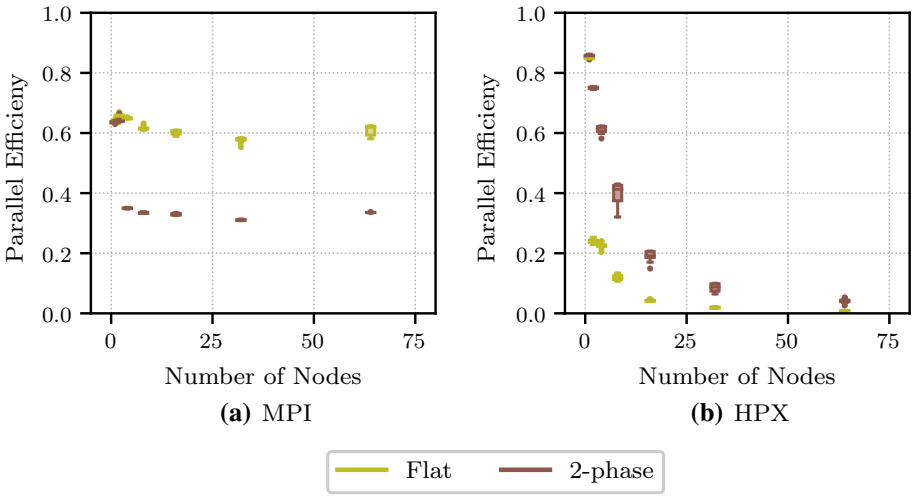


Fig. 4 Strong scaling comparison of flat and 2-phase partitioning approaches for MPI and HPX on the Knights Landing (KNL) architecture. Each data point was simulated ten times with no data being omitted

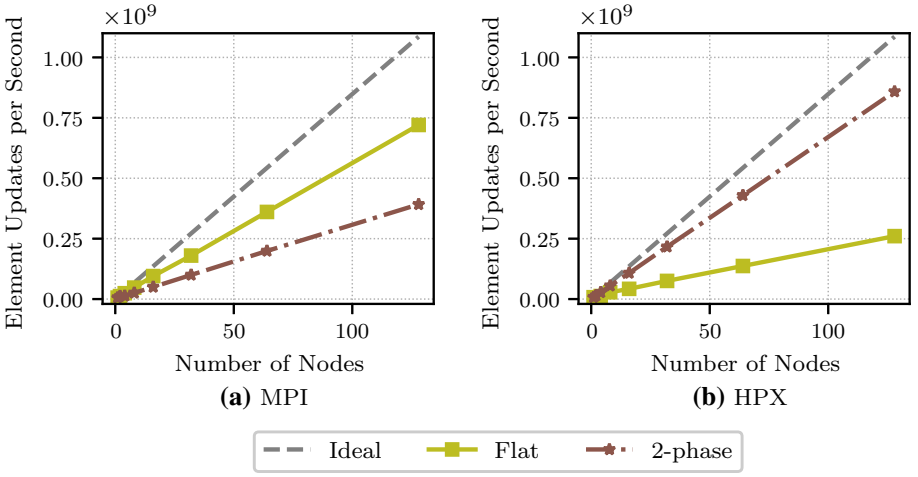


Fig. 5 Weak scaling comparison of flat and 2-phase partitioning approaches for MPI and HPX on the Knights Landing (KNL) architecture. Each data point was simulated ten times with the median value being reported here. The ideal line is the ideal weak scaling based on a serial simulation

scalable with the median parallel efficiency decreasing 2.5% between 1 node and 64 nodes for the strong scaling study, and for the weak scaling study, ${}^{w}T_{1}^{MPI} / {}^{w}T_{128}^{MPI} = 0.91$.

In contrast, the partitioning results for HPX show the 2-phase approach significantly outperforming the flat partitioning approach. For the strong scaling results shown in Fig. 4b, the 2-phase partitioning strategy is able to achieve comparable parallel efficiencies to the flat approach with as few as an eighth of the elements per submesh that the flat approach requires. Specifically, on the KNL nodes, we observe that using 2 nodes and a flat partitioning scheme we obtain a parallel efficiency of 23.6%, on the other hand, using the 2-phase approach we are able to scale out to 16 nodes, with a parallel efficiency of 19.1%. Similar to the MPI

parallelization, the 2-phase partitioner assigns non-uniform numbers of submeshes to each locality. However, since the submeshes are assigned according to their approximate computational load, the partitioner nevertheless balances the load, and aggressive work stealing by the HPX scheduler and over-subscription of submeshes ensure that cores are given sufficient work. While the flat partitioning ensures that numbers of submeshes assigned to each node is constant, the increase of inter-node communication strongly affects HPX's performance. HPX channels, which manage inter-component communication are able to optimize out overheads such as serializing messages when messages are sent to components on the same locality. For HPX, the key aspect to obtaining good performance is keeping the ratio of useful work to HPX overhead high. With the 2-phase partitioning, we find that we are able to maintain significantly improved performance over the flat partitioning scheme by minimizing this communication overhead. For the weak scaling comparison of the two partitioners shown in Fig. 5b, the 2-phase approach outperforms the flat partitioning approach by a factor of 3.30 at 128 nodes.

5 Comparison of HPX Versus MPI

We now directly compare the performance of the two parallelization approaches. In order to ensure a fair comparison, we consider the MPI and HPX parallelizations with their respective best partitioning schemes, i.e. comparing MPI runs using the flat partitioning approach and HPX runs using the 2-phase approach.

5.1 Single Node Performance Comparison

We begin by considering the weak scaling one node 1d inlet problem. The mesh contains 1,048,576 elements. To limit generated profiling data, we run this problem for a reduced time period of $t_{end} = 10$s. To profile the simulation, we use Intel's VTune Amplifier 2018 Update 2. To add profiler support to our application code, HPX is compiled with VTune Amplifier support [35], and all binaries are compiled with debugging symbols. Otherwise, we make no modifications to the application configuration outlined in Appendix A. We additionally modify the `I_MPI_FABRICS` environment variable to `shm`.

Ignoring initialization and finalization, the MPI simulation takes 67.9 s, and the HPX simulation takes 55.2 s. Both timings are within 2% of the unprofiled runtimes. The HPX parallelization provides a speed-up of 1.23 over the MPI implementation. To help explain this performance difference, we used VTune to determine how much time the respective runs were spending in each module, i.e. the application code, libraries, and kernel calls. These results are presented in Fig. 6 and the timings are shown in Table 2. The dgswemv2 module refers to the application code. Modules hpx, mkl, mpi, and jemalloc refer to time spent in these respective libraries. Lastly, vmlinux refers to time spent in the Linux kernel.

The speed-up of the HPX parallelization versus the MPI parallelization can be attributed to two main factors: firstly, the application runs 6% faster with the HPX parallelization. Due to the overdecomposition of the mesh, the HPX parallelization uses submeshes half the size of the MPI parallelization. We suspect that the performance difference arises due to better cache behavior for the smaller submeshes. This performance difference in the evaluation speed of the RK update kernels accounts for 22.6% of the overall performance difference between the MPI and HPX parallelizations. The second factor is the difference spent evaluating Linux kernel functions and accounts for 67.4% of the performance difference. Using a bottom-up profile,

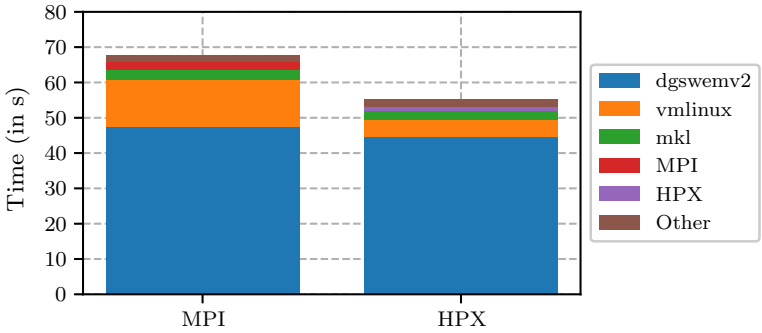


Fig. 6 Module composition of the discontinuous Galerkin method of HPX and MPI parallelizations

Table 2 Time spent in modules for single node analysis for HPX and MPI parallelization

Module	Parallelization	
	MPI	HPX
dgswemv2	47.5	44.6
vmlinux	13.3	4.8
mkl	2.7	2.5
MPI	2.5	–
HPX and jemalloc	–	1.1
Other	1.9	2.1

All timings are presented in seconds

the 5 most expensive functions are evaluated in the kernel for the MPI parallelization are in order: `native_queued_spin_lock_slowpath`, `get_page_from_freelist`, `clear_page_c_e`, `handle_mm_fault`, and `page_fault`. These functions suggest that the flat MPI parallelization generates large numbers of page faults. This overhead is not present in the HPX parallelization. These two factors account 90% of the performance discrepancy in the two approaches. Lastly, we remark that there is some imbalance, but it appears to only modestly impact application performance. The MPI implementation spends 3.9% of the time in the MPI module, including `MPI_Wait` calls. This is fairly small and partially offset by HPX runtime overhead. Thus, the performance discrepancy is caused by overhead generated by the MPI runtime, and not poor load balance or time spent waiting on messages to arrive.

5.2 Strong and Weak Scaling Studies

While the HPX parallelization outperforms the MPI parallelization in the single node analysis, the strong and weak scaling studies presented in this section explore the impact of task granularity and increased network communication.

The strong scaling results for the two approaches are shown in Fig. 7. For a single node run, HPX achieves a median parallel efficiency of 85.7%, and MPI achieves a parallel efficiency of 63.5%. This corresponds to a speed-up of 1.35 for the HPX parallelization relative to the MPI parallelization. We suspect that the MPI parallelization is incurring similar performance overheads to those noted in the previous section.

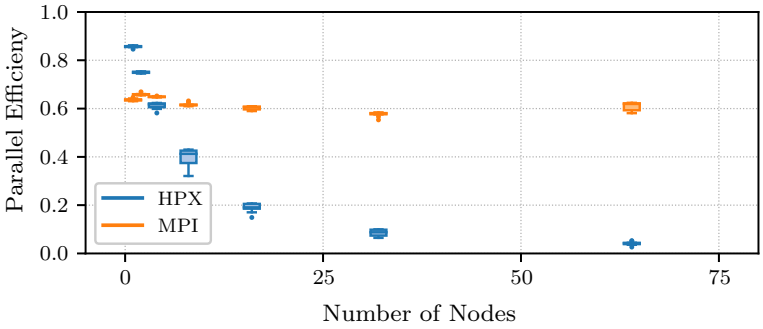


Fig. 7 Strong scaling results comparing the performance of HPX to MPI for up to 64 KNL nodes on Stampede2. For both machines, each simulation was run 10 times with no data being excluded

As we scale up in processor number, eventually the HPX task overhead begins to dominate the amount of useful work being performed, and the MPI implementation outperforms HPX. Furthermore, the scheduler's aggressive work stealing leads to factors such as false sharing further degrading performance. It is not a question of if HPX will be slower than MPI, but rather at which point. On the KNL nodes, this point occurs around the task granularity associated with 4 nodes.

To understand the performance profile of HPX's overhead, we look at the relation between the average task size versus the average overhead using HPX's performance counters. The average task size, t_{avg} is computed as

$$t_{avg} = \frac{\sum t_{app}}{n_t},$$

and the average task overhead, t_o is computed as

$$t_o = \frac{\sum t_{thread} - \sum t_{app}}{n_t},$$

where $\sum t_{app}$ is defined as the amount of time spent executing application tasks, $\sum t_{thread}$ is defined as the total execution time of all HPX threads, and n_t is the number of application threads. Both the average task size and the average task overhead are reported in units of time. The average task size is evaluated using the `/threads/time/average` counter and the average task overhead is evaluated using the `/threads/time/average-overhead` counter. For further details on the counters, we refer the reader to [25]. Table 3 and Fig. 8 show the composition of the thread execution times for the strong scaling results. Each run is done once, with the timings being thrown out if significant deviation from the median parallel efficiency reported in Fig. 7 was observed. For these results, the largest observed deviation was 1.1% from the median reported in the strong scaling runs reported in Fig. 7. The node configurations at which MPI outperforms HPX coincide with the task overheads comprising significant portions of the thread execution times.

Although the results presented here were obtained from a DG kernel for the shallow water equations, the impact of the HPX overhead can be determined by the task granularities. As such, the regimes in which HPX runs efficiently can be generalized to arbitrary stencil-type kernels. As a rule of thumb, for the KNL nodes, we recommend not scaling beyond the grain size observed at 4 nodes, which corresponds to an average task size, t_{avg} of 3.6 ms. At

Table 3 HPX thread execution composition for strong scaling runs on Knights Landing (KNL) architecture on Stampede2

Nodes	t_{avg} (in ms)	t_o (in ms)	IR (in %)
1	14.66	0.02	0.14
2	7.12	1.15	13.91
4	3.62	1.13	23.79
8	1.99	1.74	46.65
16	1.15	2.81	70.96
32	0.65	3.01	82.24
64	0.46	4.67	91.03

Both the average task size, t_{avg} and the average task overhead, t_o are reported in milliseconds. The idle rate IR is the ratio of the task overhead over the thread execution time, i.e. $IR = t_o/(t_o + t_{avg})$. The idle rate is reported as a percentage

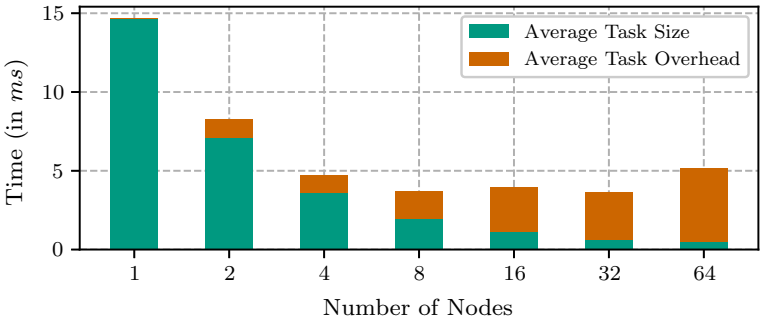


Fig. 8 Comparison of the task overhead versus the the task size for strong scaling runs for KNL nodes on Stampede2

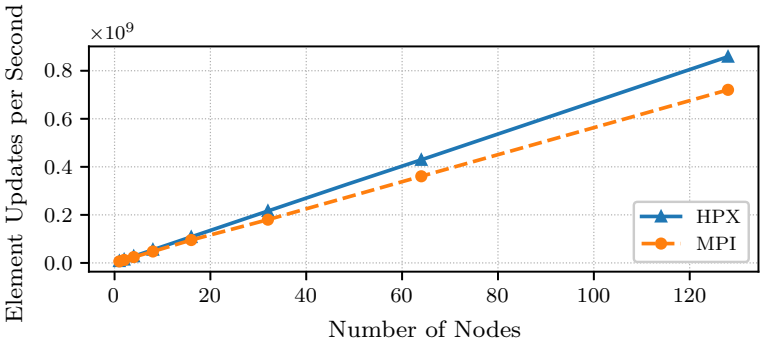


Fig. 9 Weak scaling study comparing the speed of the application to the number of nodes ranging from 1 to 128 nodes for each node type. The application speed, element updates per second is defined to be the number times any element is advanced by one Runge–Kutta stage per second. The results shown are based on the median of 10 runs

this granularity, HPX performs similarly to MPI. This task granularity is consistent with the results reported in [25].

For this weak scaling experiment, the task granularity is well within the regime where the HPX overhead constitutes a small fraction of the execution time. The results are shown

in Fig. 9. The weak scaling results show HPX outperforming MPI across the entire set of nodes in consideration. Similar to the single node analysis and strong scaling study, HPX outperforms MPI at low node counts with ${}^{w}T_{1}^{MPI}/{}^{w}T_{1}^{HPX} = 1.26$ on one node and ${}^{w}T_{2}^{MPI}/{}^{w}T_{2}^{HPX}$ decreases to 1.14 on two nodes once HPX sends messages over the interconnect. This speed-up of HPX versus MPI is maintained as we scale out to 128 nodes, with ${}^{w}T_{128}^{MPI}/{}^{w}T_{128}^{MPI} = 1.19$. These speed-ups underpin the conclusion of the previous sections that the key driver of HPX performance is the proper balancing of task overheads with useful work. By performing a weak scaling study, we are effectively fixing the scheduling overhead, and observe that HPX scales very well with ${}^{w}T_{1}^{HPX}/{}^{w}T_{128}^{HPX}$ equaling 0.86.

6 Conclusion

The massive increase in concurrency on future computer architectures necessitates the development of new programming models to efficiently utilize these architectures. In this paper, we compared the performance of an HPX versus MPI implementation of an unstructured grid DG finite element code for the shallow water equations.

Scaling results were presented for the Knights Landing processors on TACC's Stampede2. Results indicate that with a sufficiently large task size HPX is able to outperform the MPI application by a factor of approximately 1.2 with the speed-up being attributed to lower runtime overhead. However, strong HPX performance is contingent upon the task granularities remaining above a machine dependent size. For tasks with durations shorter than this machine prescribed size, overheads will dominate the execution time. For these runs, we found that MPI outperformed HPX. When considering adopting an HPX-based parallelization, the question should be "Is this performance critical task granularity sufficiently fine for my use case?" If answered in the affirmative, we believe that HPX presents a significant improvement over traditional parallelization strategies.

One of the major benefits that remains unaddressed in this paper is the ability of HPX to efficiently execute irregular applications. Whereas the kernels evaluated in this paper are load balanced statically, there exists a large class of kernels whose task size varies at run time, e.g. adaptive mesh refinement algorithms and local timestepping. These approaches are more efficient theoretically, however, statically load balanced implementations achieve a fraction of the theoretical speed-up due to resource starvation. HPX's aggressive on-node work stealing as well as task migration support should allow for more efficient implementations of these algorithms. These are topics of future work. Another application specifically related to the simulation of hurricane storm surge is load imbalance generated between the computational cost difference between wet and dry regions of the mesh. Validating the algorithms presented in [7] using HPX is another subject of ongoing work.

While this work has focused exclusively on the Knights Landing architecture, performance portability is another area of interest. The upcoming NERSC machine, Perlmutter, and the next TACC machine, Frontera, will both utilize manycore architectures similar to the Knights Landing architecture. While the task sizes that balance HPX overhead with application throughput will change, we expect that the results presented here should largely generalize to these CPU-based architectures. Targeting accelerator-based clusters with task-based programming models requires significantly more work, but is a topic of active development. HPX has support for evaluating CUDA and OpenCL kernels on GPUs. This approach allows HPX to utilize GPUs. However, HPX is not presently using task-based parallelism on the

device itself. The viability of task-based approaches on GPUs will need to navigate the trade-offs between increased asynchrony and performance.

Ultimately, applications will require new programming models to efficiently utilize exascale machines. While the results in this paper are limited in scope, we believe they are indicative of the substantial utility of a task-based approach for the next generation of computer architectures. The observed performance benefits of HPX stem from a fundamental change in the parallelization paradigm. We remark that both MPI and OpenMP are actively changing their programming models as well. In the 3.0 standard, MPI has adopted many new features such as RDMAs and shared memory operations, while OpenMP has moved from a fork-join based parallelism model to adopting task-based parallelism and adding pragmas for SIMD support. These are features that are being broadly adopted across the high performance computing community. HPX represents one solution that incorporates these concepts in a succinct and C++ standards-oriented framework, that allows developers to productively parallelize their applications.

Acknowledgements This work was supported by the National Science Foundation under Grants ACI-1339801 and ACI-1339782 and the U.S. Department of Energy through the Computational Science Graduate Fellowship, Grant DE-FG02-97ER25308. Additionally, the authors would like to acknowledge the Texas Advanced Computing Center (TACC) at the University of Texas at Austin for providing the HPC resources that enabled this research. The presented results were obtained through XSEDE allocation TG-DMS080016N.

A Artifact Description: Performance Comparison of HPX Versus Traditional Parallelization Strategies for the Discontinuous Galerkin Method

A.1 Abstract

As part of the Association for Computing Machinery's reproducibility initiative, this artifact outlines the details of generating the computational results in the paper, "Performance Comparison of HPX Versus Traditional Parallelization strategies for the Discontinuous Galerkin Method".

A.2 Description

A.2.1 Check-List (Artifact Meta Information)

- **Algorithm:** Discontinuous Galerkin Finite Element Method on Unstructured Meshes
- **Program:** C++14
- **Compilation:** GNU C++14 Compiler
- **Operating System:** CentOS Linux release 7.4.1708
- **Run-time environment:** Stampede2
- **Execution:** Parallel/Distributed
- **Experiment workflow:** Compile; Preprocess meshes; Run simulation.
- **Publicly available?:** Yes

Table 4 Version numbers or git hashes for all the software dependencies on Stampede2

Software	Stampede2
GCC	7.1.0
CMAKE	3.8.2
YAML- CPP	e2818c4
METIS	5.1.0
MKL	17.0.4
BLAZE	a292f43
MPI implementation	impi/17.0.3
HWLOC	1.11.5rc2-git
BOOST	1.64
JEMALLOC	3.6.0
HPX	b6362c2
DGSWEM- V2	7a68320

These configurations were used to generate the results shown in this paper

A.2.2 How Software Can be Obtained

DGSWEM- V2 has been released via the MIT license. The code can be obtained from the GitHub repository: https://github.com/UT-CHG/dgswemv2.

A.2.3 Software Dependencies

The building of the software stack involves several libraries. We describe the dependencies here, but provide specific version numbers in Table 4. To build the software stack, we use C++14 conforming version of the GNU C and C++ compiler. We require CMAKE to build YAML- CPP, HPX, and DGSWEM- V2. The preprocessor requires METIS to decompose the meshes, and YAML- CPP is required to parse the input files. In addition to CMAKE, HPX is also compiled with the BOOST, an MPI implementation, HWLOC, and JEMALLOC libraries.

A.3 Installation

To build DGSWEM- V2, we refer the reader to the *README.md* page at the root level of the GitHub repository. Additionally, we have made several Stampede2 specific optimizations. Firstly, we have compiled the code on the KNL architecture with the `-march=native` flag. Secondly, we have made several minor changes to optimize performance, i.e. disabling slope limiting and linking MKL. These changes have been provided as a patch file in the supplementary material.

A.4 Experiment Workflow

The workflow requires generating meshes, partitioning meshes, and executing the simulations. For brevity, we omit the details here. However, they can be found in the DGSWEM- V2*: User's Guide*, located in the `documentation/users-guide` directory of the repository.

A.5 Evaluation and Expected Result

Each simulation run will return the execution time in microseconds to the standard output.

A.6 Notes

DGSWEM- V2 is under development at the date of publication, and all new contributions can be found at https://github.com/UT-CHG/dgswemv2. If there are any comments, questions, or suggestions, we encourage communicating with the developers through the GitHub issues page.

References

1. Amarasinghe, S., Hall, M., Lethin, R., Pingali, K., Quinlan, D., Sarkar, V., Shalf, J., Lucas, R., Yelick, K., Balanji, P., et al.: Exascale programming challenges. In: Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA. US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) (2011)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **23**(2), 187–198 (2011)
3. Baggag, A., Atkins, H., Keyes, D.: Parallel implementation of the discontinuous Galerkin method. Tech. Rep. ICASE-99-35, Institute for Computer Applications in Science and Engineering (1999)
4. Balaji, P.: Programming Models for Parallel Computing. MIT Press, Cambridge (2015)
5. Barat, R.: Load balancing of multi-physics simulation by multi-criteria graph partitioning. Ph.D. thesis, Université de Bordeaux (2017)
6. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 66. IEEE Computer Society Press (2012)
7. Bremer, M.H., Bachan, J.D., Chan, C.P.: Semi-static and dynamic load balancing for asynchronous hurricane storm surge simulations. In: Proceedings of the Parallel Applications Workshop, Alternatives to MPI, p. 13. IEEE (2018)
8. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
9. Brus, S.: Efficiency improvements for modeling coastal hydrodynamics through the application of high-order discontinuous Galerkin solutions to the shallow water equations. Ph.D. thesis, University of Notre Dame (2017)
10. Brus, S.R., Wirasaet, D., Westerink, J.J., Dawson, C.: Performance and scalability improvements for discontinuous Galerkin solutions to conservation laws on unstructured grids. J. Sci. Comput. **70**(1), 210–242 (2017). https://doi.org/10.1007/s10915-016-0249-y
11. Bunya, S., Dietrich, J., Westerink, J., Ebersole, B., Smith, J., Atkinson, J., Jensen, R., Resio, D., Luettich, R., Dawson, C., Cardone, V., Cox, A., Powell, M., Westerink, H., Roberts, H.: A high resolution coupled riverine flow, tide, wind, wind wave and storm surge model for Southern Louisiana and Mississippi: part I—model development and validation. Mon. Weather Rev. **138**, 345–377 (2010)
12. Bunya, S., Kubatko, E.J., Westerink, J.J., Dawson, C.: A wetting and drying treatment for the Runge–Kutta discontinuous Galerkin solution to the shallow water equations. Comput. Methods Appl. Mech. Eng. **198**(17), 1548–1562 (2009)
13. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
14. Cockburn, B., Shu, C.W.: TVB Runge–Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework. Math. Comput. **52**(186), 411–435 (1989)
15. Cockburn, B., Shu, C.W.: Runge–Kutta discontinuous Galerkin methods for convection-dominated problems. J. Sci. Comput. **16**(3), 173–261 (2001)
16. Dawson, C., Kubatko, E.J., Westerink, J.J., Trahan, C., Mirabito, C., Michoski, C., Panda, N.: Discontinuous Galerkin methods for modeling hurricane storm surge. Adv. Water Resour. **34**(9), 1165–1176 (2011)

17. Dietrich, J., Westerink, J., Kennedy, A., Smith, J., Jensen, R.E., Zijlema, M., Holthuijsen, L., Dawson, C., Luettich, R., Powell, M., Cardone, V., Cox, A., Stone, G., Pourtaheri, H., Hope, M., Tanaka, S., Westerink, L., Westerink, H.J., Cobell, Z.: Hurricane Gustav (2008) waves and storm surge: hindcast, synoptic analysis and validation in Southern Louisiana. Mon. Weather Rev. **139**, 2488–2522 (2011)
18. Dietrich, J., Zijlema, M., Westerink, J., Holtjuijsen, L., Dawson, C., Luettich Jr., R.A., Jensen, R., Smith, J., Stelling, G., Stone, G.: Modeling hurricane wave and storm surge using integrally-coupled, scalable computations. Coast. Eng. **58**, 45–65 (2011)
19. Dietrich, J.C., Bunya, S., Westerink, J.J., Ebersole, B.A., Smith, J.M., Atkinson, J.H., Jensen, R., Resio, D.T., Luettich, R.A., Dawson, C., Cardone, V.J., Cox, A.T., Powell, M.D., Westerink, H.J., Roberts, H.J.: A high resolution coupled riverine flow, tide, wind, wind wave and storm surge model for southern Louisiana and Mississippi: part II—synoptic description and analyses of Hurricanes Katrina and Rita. Mon. Weather Rev. **138**, 378–404 (2010)
20. Dubiner, M.: Spectral methods on triangles and other domains. J. Sci. Comput. **6**(4), 345–390 (1991)
21. Dutykh, D., Clamond, D.: Modified shallow water equations for significantly varying seabeds. Appl. Math. Model. **40**(23), 9767–9787 (2016). https://doi.org/10.1016/j.apm.2016.06.033
22. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.: UPC: Distributed Shared Memory Programming, vol. 40. Wiley, London (2005)
23. Gandham, R., Medina, D., Warburton, T.: GPU accelerated discontinuous Galerkin methods for shallow water equations. Commun. Comput. Phys. **18**(1), 3764 (2015). https://doi.org/10.4208/cicp.070114.271114a
24. Gottlieb, S., Shu, C.W., Tadmor, E.: Strong stability-preserving high-order time discretization methods. SIAM Rev. **43**(1), 89–112 (2001)
25. Grubel, P., Kaiser, H., Cook, J., Serio, A.: The performance implication of task size for applications on the HPX runtime system. In: 2015 IEEE International Conference on Cluster Computing (CLUSTER), pp. 682–689. IEEE (2015)
26. Grubel, P., Kaiser, H., Huck, K.A., Cook, J.: Using intrinsic performance counters to assess efficiency in task-based parallel applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1692–1701 (2016)
27. Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R.D., Pritchard, H., Squyres, J.M.: A brief introduction to the OpenFabrics interfaces-a new network API for maximizing high performance application efficiency. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI), pp. 34–39. IEEE (2015)
28. Heller, T., Diehl, P., Byerly, Z., Biddiscombe, J., Kaiser, H.: HPX—an open source C++ standard library for parallelism and concurrency. In: Proceedings of OpenSuCo, OpenSuCo'17. ACM (2017)
29. Heller, T., Kaiser, H., Diehl, P., Fey, D., Schweitzer, M.A.: Closing the Performance Gap with Modern C++. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) High Performance Computing, *Lecture Notes in Computer Science*, vol. 9945, pp. 18–31. Springer International Publishing, Berlin (2016)
30. Heller, T., Kaiser, H., Schäfer, A., Fey, D.: Using HPX and LibGeoDecomp for scaling HPC applications on heterogeneous supercomputers. In: Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, p. 1. ACM (2013)
31. Hesthaven, J.S., Warburton, T.: Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications. Springer, Berlin (2007)
32. Hope, M.E., Westerink, J.J., Kennedy, A.B., Kerr, P.C., Dietrich, J.C., Dawson, C., Bender, C.J., et al.: Hindcast and validation of Hurricane Ike (2008) waves, forerunner, and storm surge. J. Geophys. Res. Oceans **118**, 4424–4460 (2013)
33. Hope, M., Westerink, J., Kennedy, A., Smith, J., Westerink, H., Cox, A., Nong, S., Roberts, K., Resio, D., A.P, T.: Hurricane Sandy (2012) wind, waves and storm surge in New York Bight. I: Model validation. J. Waterw. Port Coast. Ocean Eng. (2016)
34. Iglberger, K., Hager, G., Treibig, J., Rüde, U.: Expression templates revisited: a performance analysis of current methodologies. SIAM J. Sci. Comput. **34**(2), C42–C69 (2012)
35. Kaiser, H., Adelstein Lelbach, B., Heller, T., Berg, A., Biddiscombe, J., Bikineev, A., et al.: STEllAR-GROUP/hpx: HPX V1.0: The C++ standards library for parallelism and concurrency (Version 1.0.0). Zenodo. https://doi.org/10.5281/zenodo.556772 (2107)
36. Kale, L.V., Krishnan, S.: CHARM++: A portable concurrent object oriented system based on C++. In: ACM Sigplan Notices, vol. 28, pp. 91–108. ACM (1993)
37. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
38. Kogge, P., Shalf, J.: Exascale computing trends: adjusting to the "new normal" for computer architecture. Comput. Sci. Eng. **15**(6), 16–26 (2013)

39. Kubatko, E., Bunya, S., Dawson, C., Westerink, J.: Dynamic p-adaptive Runge–Kutta discontinuous Galerkin methods for the shallow water equations. Comput. Methods Appl. Mech. Eng. **198**, 1766–1774 (2009)
40. Kubatko, E., Bunya, S., Dawson, C., Westerink, J., Mirabito, C.: A performance comparison of continuous and discontinuous finite element shallow water models. J. Sci. Comput. **40**, 315–339 (2009)
41. Kubatko, E., Westerink, J., Dawson, C.: *hp* Discontinuous Galerkin methods for advection dominated problems in shallow water flow. Comput. Methods Appl. Mech. Eng. **196**, 437–451 (2006)
42. Kubatko, E.J., Westerink, J.J., Dawson, C.: Semi discrete discontinuous Galerkin methods and stage-exceeding-order, strong-stability-preserving Runge–Kutta time discretizations. J. Comput. Phys. **222**(2), 832–848 (2007)
43. Luettich, R., et al. J.W.: ADCIRC: a parallel advanced circulation model for oceanic, coastal and estuarine waters (2017). Users manual www.adcirc.org
44. Michoski, C., Alexanderian, A., Paillet, C., Kubatko, E., Dawson, C.: Stability of nonlinear convection-diffusion-reaction systems in discontinuous Galerkin methods. J. Sci. Comput. **70**, 516–550 (2017)
45. Michoski, C., Dawson, C., Kubatko, E., Wirasaet, D., Brus, S., Westerink, J.: A comparison of artificial viscosity, limiters, and filter, for high order discontinuous Galerkin solution in nonlinear settings. J. Sci. Comput. (2015). https://doi.org/10.1007/s10915.015.0027.2
46. Michoski, C., Dawson, C., Mirabito, C., Kubatko, E., Wirasaet, D., Westerink, J.: Fully coupled methods for multiphase morphodynamics. Adv. Water Resour. **59**, 95–110 (2013)
47. Michoski, C., Mirabito, C., Dawson, C., Wirasaet, D., Kubatko, E.J., Westerink, J.J.: Adaptive hierarchic transformations for dynamically p-enriched slope-limiting over discontinuous Galerkin systems of generalized equations. J. Comput. Phys. **230**(22), 8028–8056 (2011)
48. Numrich, R.W., Reid, J.: Co-Array Fortran for parallel programming. In: ACM Sigplan Fortran Forum, vol. 17, pp. 1–31. ACM (1998)
49. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0 (2008). http://www.openmp.org/mp-documents/spec30.pdf
50. Reed, W.H., Hill, T.: Triangular mesh methods for the neutron transport equation. Tech. rep., Los Alamos Scientific Lab., N. Mex. (USA) (1973)
51. Tanaka, S., Bunya, S., Westerink, J., Dawson, C., Luettich, R.: Scalability of an unstructured grid continuous Galerkin based hurricane storm surge model. J. Sci. Comput. **46**, 329–358 (2011)
52. Westerink, J.J., Luettich, R.A., Feyen, J.C., Atkinson, J.H., Dawson, C.N., Roberts, H.J., Powell, M.D., Dunion, J.P., Kubatko, E.J., Pourtaheri, H.: A basin to channel scale unstructured grid hurricane storm surge model applied to southern Louisiana. Mon. Weather Rev. **136**, 833–864 (2008)
53. Wirasaet, D., Brus, S., Michoski, C., Kubatko, E., Westerink, J.: Artificial boundary layers in discontinuous Galerkin solutions to shallow water equations in channels. J. Comput. Physics **299**, 597–612 (2015)
54. Wirasaet, D., Kubatko, E., Michoski, C., Tanaka, S., Westerink, J., Dawson, C.: Discontinuous Galerkin methods with nodal and hybrid modal/nodal triangular, quadrilateral, and polygonal elements for nonlinear shallow water flow. Comput. Methods Appl. Mech. Eng. **270**, 113–149 (2014)