An Implicitly Parallel Meshfree Solver in Regent

Rupanshu Soi
Department of Computer Science
BITS Pilani - Hyderabad Campus
Hyderabad, India
f20180294@hyderabad.bits-pilani.ac.in

Nischay Ram Mamidi
Department of Mathematics
BITS Pilani - Hyderabad Campus
Hyderabad, India
nischay@hyderabad.bits-pilani.ac.in

Elliott Slaughter
Computer Science Research Department
SLAC National Accelerator Laboratory
Menlo Park, USA
eslaught@slac.stanford.edu

Kumar Prasun
Department of Computer Science
BITS Pilani - Hyderabad Campus
Hyderabad, India
f20150845@hyderabad.bits-pilani.ac.in

Anil Nemili
Department of Mathematics
BITS Pilani - Hyderabad Campus
Hyderabad, India
anil@hyderabad.bits-pilani.ac.in

S.M. Deshpande
Engineering Mechanics Unit
JNCASR
Bangalore, India
smd@jncasr.ac.in

***Abstract*—This paper presents the development of a Regent based implicitly parallel meshfree solver for inviscid compressible fluid flows. The meshfree solver is based on the Least Squares Kinetic Upwind Method (LSKUM). The performance of the Regent parallel solver is assessed by comparing with the explicitly parallel versions of the same solver written in Fortran 90 and Julia. The Fortran code uses MPI with PETSc libraries, while the Julia code uses an MPI + X alternative parallel library. Numerical results are shown to assess the performance of these solvers on single and multiple CPU nodes.**

***Index Terms*—Regent, Legion, Fortran, Julia, MPI+X, LSKUM, Meshfree methods**

I. Introduction

Modern high performance computing platforms have become highly heterogeneous as the computing devices are composed of multi-core CPUs and GPUs with varying specifications. It is well-known that the numerical simulation of many complex fluid flow problems require massively parallel computational facilities. Systems which can asynchronously schedule the simultaneous execution of tasks on both CPUs and GPUs are described in the literature [1]–[3], but have not yet seen broad adoption in large-scale HPC applications. For example, the well-known CFD codes such as SU2 [4] uses CPUs while OpenFOAM [5] and PyFR [6] can run on either CPUs or GPUs, but not both simultaneously. Furthermore, it will be of great advantage if the code is based on implicit parallelism as it significantly enhances readability, portability and productivity. The Regent programming language [7] precisely addresses these concerns.

Regent is a high level, task-based programming language that presents a viable alternative to traditional distributed programming through its support for implicit parallelism. Programs consist of *tasks* or functions marked as being eligible for parallel execution. Tasks appear to the programmer as if they execute in sequential, program order; the programming system is responsible for the extraction of parallelism. Regent also lets users run the same code on vastly different machine architectures with virtually zero application changes. Regent's optimizing compiler together with the underlying task-based Legion runtime system [1] is able to produce programs that are able to achieve performance on par with traditional explicitly parallel applications while keeping the code easier to read, write, debug and maintain.

The long term aim of our research is to develop a three-dimensional hybrid CFD solver that is capable of exploiting the full computing power of heterogeneous platforms. Towards this objective, in the present work we have developed a two-dimensional implicitly parallel solver in a meshfree framework [8] for inviscid compressible fluid flows.

This paper is organised as follows. Section II presents the basic theory of the meshfree solver based on the least squares kinetic upwind method. Section III discusses details pertaining to the construction of a Regent based meshfree solver. In Section IV, numerical results are shown to analyse the computational performance of the solver. Finally, conclusions are drawn in Section V.

II. The Kinetic Meshfree Method: q-LSKUM

A. Basic theory of LSKUM

The Least Squares Kinetic Upwind Method (LSKUM) [9] belongs to the family of kinetic theory based upwind schemes for the numerical solution of Euler or Navier-Stokes equations that govern the compressible fluid flows. These schemes are based on the moment-method-strategy [10], where an upwind scheme is first developed at the Boltzmann level and after taking suitable moments, we arrive at an upwind scheme for the governing conservation laws. LSKUM requires a distribution of points, which can be structured or unstructured. The point distributions can be obtained from simple or chimera point generation algorithms, quadtree, or even advancing front methods [11]. In this section, we briefly present the basic theory of LSKUM for two-dimensional $(2D)$ Euler equations that govern the inviscid compressible fluid flows.

In the differential form, the Euler equations in two-dimensions are given by

$$\frac{\partial \boldsymbol{U}}{\partial t}+\frac{\partial \boldsymbol{G}}{\partial x}+\frac{\partial \boldsymbol{H}}{\partial y}=0 \tag{1}$$

Here, $\boldsymbol{U}$ is the conserved vector, $\boldsymbol{G}$ and $\boldsymbol{H}$ are the flux vectors along the coordinate directions x and y respectively. The conservation laws in eq. (1) can be obtained by taking $\boldsymbol{\Psi}$ - moments of the $2D$ Boltzmann equation in the Euler limit. In the inner product form, this can be written as

$$\frac{\partial \boldsymbol{U}}{\partial t}+\frac{\partial \boldsymbol{G}}{\partial x}+\frac{\partial \boldsymbol{H}}{\partial y}=\left\langle \boldsymbol{\Psi}, \frac{\partial F}{\partial t}+v_1\frac{\partial F}{\partial x}+v_2\frac{\partial F}{\partial y}\right\rangle=0 \tag{2}$$

Here, F is the Maxwellian velocity distribution function and $\boldsymbol{\Psi}$ is the moment function vector. v_1 and v_2 are the molecular velocities along the coordinate directions x and y respectively. The inner product $\langle \boldsymbol{\Psi}, f\rangle$ is defined as

$$\langle \boldsymbol{\Psi}, f\rangle=\int\limits_{\mathbb{R}^+\times\mathbb{R}^2} \boldsymbol{\Psi} f\left(\boldsymbol{v}\right) d\boldsymbol{v} dI \tag{3}$$

Using Courant-Issacson-Rees (CIR) splitting [12] of molecular velocities, an upwind scheme for the Boltzmann equation in eq. (2) can be constructed as

$$\frac{\partial F}{\partial t}+v_1^+\frac{\partial F}{\partial x}+v_1^-\frac{\partial F}{\partial x}+v_2^+\frac{\partial F}{\partial y}+v_2^-\frac{\partial F}{\partial y}=0 \tag{4}$$

where, the split velocities $v_1^\pm$ and $v_2^\pm$ are defined as

$$v_1^\pm=\frac{v_1\pm|v_1|}{2},\ \ v_2^\pm=\frac{v_2\pm|v_2|}{2} \tag{5}$$

The basic idea of LSKUM is to obtain discrete approximations to the spatial derivatives using least squares principle. We illustrate this approach to determine F_x and F_y at a point P_0 using the data at its neighbours. The set of neighbours, also known as the stencil of P_0 is defined by

$$N\left(P_0\right)=\left\{P_i : d\left(P_0,P_i\right)<\epsilon\right\} \tag{6}$$

where $d\left(P_i,P_0\right)$ is the Euclidean distance between the points P_0 and P_i. ϵ is the user defined characteristic linear dimension of $N\left(P_0\right)$.

To derive the least squares approximation of F_x and F_y, consider the Taylor series expansion of F up to linear terms at a neighbour point P_i around P_0,

$$\Delta F_i=\Delta x_i F_{x0}+\Delta y_i F_{y0}+O\left(\Delta x_i,\Delta y_i\right)^2, i=1,\ldots,n \tag{7}$$

where $\Delta x_i=x_i-x_0$, $\Delta y_i=y_i-y_0$, $\Delta F_i=F_i-F_0$ and n represents the number of neighbours of the point P_0. For $n\geq 3$, eq. (7) leads to an over-determined linear system, which can be solved using the least squares principle. The first-order accurate least squares approximations to the partial derivatives F_x and F_y at the point P_0 are then given by

$$\begin{bmatrix} F_x^{\mathit{1}} \\ F_y^{\mathit{1}}\end{bmatrix}=\begin{bmatrix}\sum\Delta x_i^2 & \sum\Delta x_i\Delta y_i\\ \sum\Delta x_i\Delta y_i & \sum\Delta y_i^2\end{bmatrix}^{-1}\begin{bmatrix}\sum\Delta x_i\Delta F_i\\ \sum\Delta y_i\Delta F_i\end{bmatrix} \tag{8}$$

In the above formulae, the superscript *1* on F_x and F_y denotes first-order accuracy. Taking $\boldsymbol{\Psi}$ - moments of eq. (4) along with the formulae in eq. (8), we obtain the semi-discrete form of the first-order least squares kinetic upwind scheme for $2D$ Euler equations,

$$\frac{\partial \boldsymbol{U}}{\partial t}+\frac{\partial \boldsymbol{G}^+}{\partial x}+\frac{\partial \boldsymbol{G}^-}{\partial x}+\frac{\partial \boldsymbol{H}^+}{\partial y}+\frac{\partial \boldsymbol{H}^-}{\partial y}=0 \tag{9}$$

Here, $\boldsymbol{G}^\pm$ and $\boldsymbol{H}^\pm$ are respectively the kinetic split fluxes [8] along x and y directions. The least squares formulae for the split flux derivatives $\boldsymbol{G}_x^\pm$ and $\boldsymbol{H}_y^\pm$ are given by

$$\begin{aligned}\boldsymbol{G}_x^\pm&=\left[\frac{\sum\Delta y_i^2\sum\Delta x_i\Delta\boldsymbol{G}_i^\pm-\sum\Delta x_i\Delta y_i\sum\Delta y_i\Delta\boldsymbol{G}_i^\pm}{\sum\Delta x_i^2\sum\Delta y_i^2-\sum\Delta x_i\Delta y_i}\right]\\ \boldsymbol{H}_y^\pm&=\left[\frac{\sum\Delta x_i^2\sum\Delta y_i\Delta\boldsymbol{H}_i^\pm-\sum\Delta x_i\Delta y_i\sum\Delta x_i\Delta\boldsymbol{H}_i^\pm}{\sum\Delta x_i^2\sum\Delta y_i^2-\sum\Delta x_i\Delta y_i}\right]\end{aligned} \tag{10}$$

Note that $\boldsymbol{G}_x^\pm$ and $\boldsymbol{H}_y^\pm$ are evaluated using the split stencils $N_x^\pm\left(P_0\right)$ and $N_y^\pm\left(P_0\right)$ respectively. These subsets are defined by

$$\begin{aligned}N_x^\pm\left(P_0\right)&=\left\{P_i \mid P_i\in N\left(P_0\right), \Delta x_i=x_i-x_0\lesseqgtr 0\right\}\\ N_y^\pm\left(P_0\right)&=\left\{P_i \mid P_i\in N\left(P_0\right), \Delta y_i=y_i-y_0\lesseqgtr 0\right\}\end{aligned} \tag{11}$$

B. Second-order accuracy using $\mathbf{q}$*-variables*

An efficient way of obtaining second-order accurate approximations to the spatial derivatives F_x and F_y is by employing the defect correction method. An advantage of this approach is that the dimension of the least squares matrix remains the same as in the first-order scheme. To derive the desired formulae, consider the Taylor expansion of F up to quadratic terms,

$$\begin{aligned}\Delta F_i=&\Delta x_i F_{x_0}+\Delta y_i F_{y_0}+\frac{\Delta x_i}{2}\left(\Delta x_i F_{xx_0}+\Delta y_i F_{xy_0}\right)\\ &+\frac{\Delta y_i}{2}\left(\Delta x_i F_{xy_0}+\Delta y_i F_{yy_0}\right)\\ &+O\left(\Delta x_i,\Delta y_i\right)^3,\ \ i=1,\ldots,n\end{aligned} \tag{12}$$

The basic idea of the defect correction procedure is to cancel the second-order derivative terms in the above equation by defining a modified ΔF_i so that the leading terms in the truncation errors of the formulae for F_x and F_y are of the order of $O\left(\Delta x_i,\Delta y_i\right)^2$. Towards this objective, consider the Taylor series expansions of F_x and F_y up to linear terms

$$\begin{aligned}\Delta F_{x_i}=&\Delta x_i F_{xx_0}+\Delta y_i F_{xy_0}+O\left(\Delta x_i,\Delta y_i\right)^2\\ \Delta F_{y_i}=&\Delta x_i F_{xy_0}+\Delta y_i F_{yy_0}+O\left(\Delta x_i,\Delta y_i\right)^2\end{aligned} \tag{13}$$

where $\Delta F_{x_i}=F_{x_i}-F_{x_0}$ and $\Delta F_{y_i}=F_{y_i}-F_{y_0}$. Using these expressions in eq. (12), we obtain

$$\begin{aligned}\Delta F_i=&\Delta x_i F_{x_0}+\Delta y_i F_{y_0}+\frac{1}{2}\Delta x_i\Delta F_{x_i}+\frac{1}{2}\Delta y_i\Delta F_{y_i}\\ &+O\left(\Delta x_i,\Delta y_i\right)^3,\ \ i=1,\ldots,n\end{aligned} \tag{14}$$

We now introduce the modified perturbation in Maxwellians, $\Delta\widetilde{F}_i$ and define it as

$$\begin{aligned}\Delta\widetilde{F}_i=&\Delta F_i-\frac{1}{2}\Delta x_i\Delta F_{x_i}-\frac{1}{2}\Delta y_i\Delta F_{y_i}\\ =&\Delta F_i-\frac{1}{2}\Delta x_i\left(F_{x_i}-F_{x_0}\right)-\frac{1}{2}\Delta y_i\left(F_{y_i}-F_{y_0}\right)\end{aligned} \tag{15}$$

Using $\Delta\widetilde{F}_i$, eq. (14) reduces to

$$\Delta\widetilde{F}_i = \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + O\left(\Delta x_i, \Delta y_i\right)^3, \quad i = 1, \ldots, n \tag{16}$$

Solving the modified over-determined system using least squares, the second-order accurate approximations to F_x and F_y at the point P_0 are given by

$$\begin{bmatrix} F_x^{2} \\ F_y^{2} \end{bmatrix} = \begin{bmatrix} \sum \Delta x_i^2 & \sum \Delta x_i \Delta y_i \\ \sum \Delta x_i \Delta y_i & \sum \Delta y_i^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum \Delta x_i \Delta \widetilde{F}_i \\ \sum \Delta y_i \Delta \widetilde{F}_i \end{bmatrix} \tag{17}$$

Note that the superscript *2* on F_x and F_y denotes second-order accuracy. The above formulae satisfy the test of k-exactness as they yield exact derivatives for polynomials of degree ≤ 2. Furthermore, these formulae have the same structure as the first-order formulae in eq. (8), except that the second-order approximations use modified Maxwellians. In contrast to first-order formulae that are explicit in nature, the second-order approximations have implicit dependence as the evaluation of F_x and F_y at the point P_0 requires the values of F_x and F_y at P_0 and its neighbours a priori, so that $\Delta\widetilde{F}_i$ in eq. (15) can be estimated.

Taking $\mathbf{\Psi}$ - moments of the spatial terms in eq. (4) along with the formulae in eq. (17), we get the second-order accurate discrete approximations for the kinetic split flux derivatives as

$$\begin{aligned} \frac{\partial \boldsymbol{G}^{\pm}}{\partial x} &= \frac{\sum \Delta y_i^2 \sum \Delta x_i \Delta \widetilde{\boldsymbol{G}}_i^{\pm} - \sum \Delta x_i \Delta y_i \sum \Delta y_i \Delta \widetilde{\boldsymbol{G}}_i^{\pm}}{\sum \Delta x_i^2 \sum \Delta y_i^2 - \sum \Delta x_i \Delta y_i} \\ \frac{\partial \boldsymbol{H}^{\pm}}{\partial y} &= \frac{\sum \Delta x_i^2 \sum \Delta y_i \Delta \widetilde{\boldsymbol{H}}_i^{\pm} - \sum \Delta x_i \Delta y_i \sum \Delta x_i \Delta \widetilde{\boldsymbol{H}}_i^{\pm}}{\sum \Delta x_i^2 \sum \Delta y_i^2 - \sum \Delta x_i \Delta y_i} \end{aligned} \tag{18}$$

where, the perturbations $\Delta\widetilde{\boldsymbol{G}}_i^{\pm}$ and $\Delta\widetilde{\boldsymbol{H}}_i^{\pm}$ are defined by

$$\begin{aligned} \Delta\widetilde{\boldsymbol{G}}_i^{\pm} &= \Delta \boldsymbol{G}_i^{\pm} - \frac{1}{2}\left\{\Delta x_i \frac{\partial}{\partial x}\Delta \boldsymbol{G}_i^{\pm} + \Delta y_i \frac{\partial}{\partial y}\Delta \boldsymbol{G}_i^{\pm}\right\} \\ \Delta\widetilde{\boldsymbol{H}}_i^{\pm} &= \Delta \boldsymbol{H}_i^{\pm} - \frac{1}{2}\left\{\Delta x_i \frac{\partial}{\partial x}\Delta \boldsymbol{H}_i^{\pm} + \Delta y_i \frac{\partial}{\partial y}\Delta \boldsymbol{H}_i^{\pm}\right\} \end{aligned} \tag{19}$$

A drawback of this formulation is that the second-order scheme thus obtained reduces to first-order at the boundaries as the stencils to compute the split flux derivatives may not have enough neighbours. Furthermore, $\Delta\widetilde{F}_i$ is not the difference between two Maxwellians. Instead, it is the difference between two perturbed Maxwellians, given by

$$\begin{aligned} \Delta\widetilde{F}_i = \widetilde{F}_i - \widetilde{F}_0 = &\left\{F_i - \frac{1}{2}\left(\Delta x_i F_{x_i} + \Delta y_i F_{y_i}\right)\right\} \\ &- \left\{F_0 - \frac{1}{2}\left(\Delta x_i F_{x_0} + \Delta y_i F_{y_0}\right)\right\} \end{aligned} \tag{20}$$

Unlike F_i and F_0, the distribution functions $\widetilde{F}_i$ and $\widetilde{F}_0$ may not be non-negative and therefore need not be Maxwellians.

In order to preserve positivity, instead of Maxwellians, we employ the $\boldsymbol{q}$-variables [13], [14] in the defect correction procedure to obtain second-order accuracy. Note that the transformations $F \longleftrightarrow \boldsymbol{q}$ and $\boldsymbol{U} \longleftrightarrow \boldsymbol{q}$ are unique and therefore the $\boldsymbol{q}$-variables can be used to represent the fluid flow at the macroscopic level. The second-order LSKUM based on $\boldsymbol{q}$-variables is then obtained by replacing $\Delta\widetilde{\boldsymbol{G}}_i^{\pm}$ and $\Delta\widetilde{\boldsymbol{H}}_i^{\pm}$ in eq. (19) with $\Delta \boldsymbol{G}_i^{\pm}\left(\widetilde{\boldsymbol{q}}\right)$ and $\Delta \boldsymbol{H}_i^{\pm}\left(\widetilde{\boldsymbol{q}}\right)$ respectively. The new perturbations in split fluxes are defined by

$$\begin{aligned} \Delta \boldsymbol{G}_i^{\pm}\left(\widetilde{\boldsymbol{q}}\right) &= \boldsymbol{G}^{\pm}\left(\widetilde{\boldsymbol{q}}_i\right) - \boldsymbol{G}^{\pm}\left(\widetilde{\boldsymbol{q}}_0\right) \\ \Delta \boldsymbol{H}_i^{\pm}\left(\widetilde{\boldsymbol{q}}\right) &= \boldsymbol{H}^{\pm}\left(\widetilde{\boldsymbol{q}}_i\right) - \boldsymbol{H}^{\pm}\left(\widetilde{\boldsymbol{q}}_0\right) \end{aligned} \tag{21}$$

Here, $\widetilde{\boldsymbol{q}}_i$ and $\widetilde{\boldsymbol{q}}_0$ are the modified $\boldsymbol{q}$-variables, given by

$$\begin{aligned} \widetilde{\boldsymbol{q}}_i &= \boldsymbol{q}_i - \frac{1}{2}\left(\Delta x_i \boldsymbol{q}_{xi} + \Delta y_i \boldsymbol{q}_{y_i}\right) \\ \widetilde{\boldsymbol{q}}_0 &= \boldsymbol{q}_0 - \frac{1}{2}\left(\Delta x_i \boldsymbol{q}_{x0} + \Delta y_i \boldsymbol{q}_{y_0}\right) \end{aligned} \tag{22}$$

The necessary condition for obtaining second-order accurate split flux derivatives is that both $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ in eq. (22) should be second-order. Note that these components are approximated using full stencil in a way similar to that of F_x and F_y in eq. (17). The resulting least squares formulae for $\boldsymbol{q}_x^{2}$ and $\boldsymbol{q}_y^{2}$ are implicit in nature and need to be solved iteratively. These sub-iterations are called inner iterations. In the present work, numerical simulations are performed with 3 inner iterations. Once evaluated, the $\widetilde{\boldsymbol{q}}$-variables are used to compute $\Delta \boldsymbol{G}^{\pm}\left(\widetilde{\boldsymbol{q}}\right)$ and $\Delta \boldsymbol{H}^{\pm}\left(\widetilde{\boldsymbol{q}}\right)$ at the first step and then the split flux derivatives in eq. (18).

An advantage of this approach is that higher-order accuracy can be achieved even at boundary points as $\boldsymbol{q}$-variables can be combined with the kinetic wall [10] and outer boundary [15] conditions. Furthermore, the distribution functions $F\left(\widetilde{\boldsymbol{q}}_i\right)$ and $F\left(\widetilde{\boldsymbol{q}}_0\right)$ corresponding to $\widetilde{\boldsymbol{q}}_i$ and $\widetilde{\boldsymbol{q}}_0$ are always Maxwellians and therefore preserves the positivity of numerical solution.

Finally, the state-update formula for steady problems can be constructed by replacing the pseudo-time derivative in eq. (9) with a suitable discrete approximation and local time stepping. In the present work, the solution is updated using a four-stage Runge-Kutta (SSP-RK3) [16] time marching algorithm.

III. Implicitly Parallel Meshfree Solver

In this section, we present the development and implementation of an implicitly parallel meshfree solver based on Regent. Typically, the development of a parallel CFD solver involves two important steps, namely, domain decomposition and data communication and synchronization between different partitions. We will consider both aspects of the Regent implementation and draw comparisons with Fortran and Julia.

Let us first consider the general outline of the solver, presented in Algorithm 1, in context of the theory presented in Section II. Each time iteration of the numerical scheme involves the computation of local time step, four stages of the Runge-Kutta scheme and L_2 norm of the residue. The subroutine `q_variables()` evaluates the $\boldsymbol{q}$-variables, while `q_derivatives()` computes the second order accurate approximations of $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ along with inner iterations. The most time consuming routine is the `flux_residual()`, which computes the sum of split flux derivatives in eq. (9). The flow solution at each Runge-Kutta step is updated in `state_update(rk)`. The domain decomposition is performed in `preprocessor()` while `postprocessor()` includes all output operations. N represents the number of pseudo-time iterations required to achieve a desired convergence in the solution.

Algorithm 1: Meshfree solver based on q-LSKUM

```
subroutine q-LSKUM
    call preprocessor()
    for $n \leftarrow 1$ to $n \leq N$ do
        call timestep()
        for $rk \leftarrow 1$ to $4$ do
            call q_variables()
            call q_derivatives()
            call flux_residual()
            call state_update(rk)
        end
        call residue()
    end
    call postprocessor()
end subroutine
```

To understand how domain decomposition is done in the Regent implementation, we must first understand how the input point distribution is stored and managed. Regent's data model consists of storing data in *regions*. To a first approximation, a region is just an array of structs. Unlike traditional arrays, regions can be relocated (e.g., onto GPUs) or can even have multiple copies at once (e.g., for replication of read-only data). Regions, when used as arguments to tasks, are also analyzed by the compiler and runtime to extract task and data parallelism. The present solver stores the input point distribution in a 1 dimensional region with multiple attributes corresponding to a point grouped together inside a C-like struct called `DomainPt`. Since this solver uses meshfree methods on an unstructured grid, each `DomainPt` also needs to store the global indices of its neighbors.

Domain decomposition is done by partitioning the parent region into *subregions*. A *partition*, defined on a region, is a first-class data type in Regent that divides the parent region into an array of subregions. If it is guaranteed that no two subregions of a partition can share a common element then the partition is said to be *disjoint*, otherwise it is *aliased*. Partitioning is a fairly cheap operation in Regent because memory is only allocated for the subregions of a partition when needed (lazy allocation). Since the amount of data parallelism in an application directly depends on how partitions have been defined, Regent provides an expressive framework for partitioning [17] which has been extremely useful in the current application.

Two different partitioning schemes were tried for the current application. First, we uniformly divided the parent region into the required number of subregions, without regards to the distribution of points within each subregion. This approach can be easily implemented in Regent. Later it was found that using METIS [18] to create a minimal edge-cut partition and using it inside Regent led to better memory usage and performance. This is due to the fact that METIS is able to minimize the total number of ghost points which reduces the amount of data communication between nodes. In Listing 2, `p_local` is the Regent partition that was created directly from METIS, which means that it is known at compile time that `p_local` is a disjoint partition over the entire input point distribution.

One difference between the three implementations is in how ghost points were determined. In both approaches outlined above, ghost points for a given partition were determined completely inside Regent. One method for this is to first connect each point to its neighbors by using a directed edge data structure. Then, any such neighbor assigned to a different subregion in `p_local` can easily be marked as a ghost point. All ghost points are then collected inside a Regent partition `p_ghost` (see line 1 in Listing 2) for easy access. Put differently, `p_ghost` is an aliased Regent partition such that the subregion `p_ghost[i]` will contain every ghost point for all points in `p_local[i]`. Note that since partitioning in Regent only consists of naming subsets of data, no data is copied between the two partitions at this time; copies will only be made as and when required during execution. On the other hand, Fortran or Julia's programming models are not expressive enough to identify ghost points automatically so all the necessary information (e.g., sizes of halos, and their connectivity) must be computed by hand from the output of METIS.

Recall that each `DomainPt` stores the global indices of its neighbors. Due to the unstructured nature of the present application domain, there is no way to tell beforehand if a given neighbor is a ghost point or not, that is, if a given neighbor is present in the same subregion or not. Fortran and Julia handle this by keeping a flag and checking it to determine whether a neighbor is a local point or not. This introduces a branch inside performance critical code. A better solution is possible in Regent. Since Regent allows us to perform the union of two partitions, we are able to create a single partition of neighbors `p_nbhs` that includes both local and ghost points. This is done in line 1 in Listing 2 for all subregions at once. Therefore, any neighbor of a point in `p_local[i]` can be accessed simply by indexing the subregion `p_nbhs[i]` by the neighbor's global index, removing the need to check a flag for each neighbor access in Regent. Note that any data communication and synchronization required to access a point belonging to a different subregion is completely hidden from the user and handled automatically by the programming system. We will look at this in more detail later.

Let us now consider the implementation of the subroutines from Algorithm 1 and understand how parallelism is extracted in Regent and Legion. First, recall from the introduction that Regent has a task-based programming model. Each task that receives a region as an input parameter must declare its *privileges* on it at compile time. A privilege can be one of `reads`, `writes` or `reads writes`. As an illustration, Listing 1 shows the task for `state_update(rk)`. It requests read and write privileges on different members of DomainPt, which is contained in the input *region requirement*. Privilege declaration is crucial to Regent's extraction of parallelism, because it lets the compiler determine which tasks are *non-interfering* (i.e., can be run in parallel). For instance, the compiler recognizes that two sibling tasks requesting read privileges on the same region cannot have a dependence among them, and are therefore safe to run concurrently. The ispace(int1d) expression verifies that the region requirement is a 1 dimensional ordered collection, as mentioned earlier. Note that the loop on lines 7–9 will be executed sequentially, since only inter-task parallelism is extracted in *pure* Regent.

```
 1 task state_update(r : region(ispace(int1d),
       DomainPt), rk : int)
 2 where
 3   reads(r.{nx, ny, prim, flux_res}),
 4   writes(r.prim)
 5 do
 6   var sum_res_sqr = 0.0
 7   for pt in r do
 8     -- computations
 9   end
10   return sum_res_sqr
11 end
```

Listing 1. Regent code for state_update().

```
 1 var p_nbhs = p_local | p_ghost
 2 for i = 0, N do
 3   __demand(__index_launch)
 4   for color in p_local.colors do
 5     time_step(p_local[color])
 6   end
 7   var res : double = 0.0
 8   for rk = 1, 5 do
 9     __demand(__index_launch)
10     for color in p_local.colors do
11       q_variables(p_local[color])
12     end
13     __demand(__index_launch)
14     for color in p_local.colors do
15       q_derivatives(p_local[color], p_nbhs[color])
16     end
17     __demand(__index_launch)
18     for color in p_local.colors do
19       flux_residual(p_local[color], p_nbhs[color])
20     end
21     __demand(__index_launch)
22     for color in p_local.colors do
23       res += state_update(p_local[color], rk)
24     end
25   end
26   residue(res)
27 end
```

Listing 2. Regent task for q-LSKUM.

Listing 2 presents Regent code for the four stage Runge-Kutta scheme. Notice that instead of calling each task on the entire point distribution at once, we iterate over our main partition p_local and call each task on each subregion. This is a common idiom in Regent that helps to expose parallelism in an application to the Regent compiler and the Legion runtime system. For instance, consider the loop on lines 10-12 in Listing 2. Notice that a call to q_variables() only receives a subregion of p_local as input. Because it is known that p_local is disjoint partition at compile time, the compiler is able to prove that, in a given time-step, no two calls to q_variables() will be interfering, and thus the loop is safe to be parallelized.

We should emphasize here that the extraction of parallelism does not stop with the Regent compiler. During execution, Legion dynamically creates a dependence graph between tasks and uses it to extract even more parallelism from the application. This is beneficial as the compiler is often forced to make conservative decisions due to the lack of information available at compile time. However, the original sequential semantics of the application are guaranteed to be maintained in every execution, distributed or otherwise.

```
subroutine update_begin_prim_ghost()
  implicit none
  PetscErrorCode      :: ierr
  if (proc==1) return

  call VecGhostUpdateBegin(p_prim,INSERT_VALUES,
    SCATTER_FORWARD,ierr)
end subroutine

subroutine update_end_prim_ghost()
  implicit none
  PetscErrorCode      :: ierr
  if (proc==1) return

  call PetscLogEventBegin(prim_comm, ierr)
  call VecGhostUpdateEnd(p_prim,INSERT_VALUES,
    SCATTER_FORWARD,ierr)
  call PetscLogEventEnd(prim_comm, ierr)
end subroutine
```

Listing 3. Fortran subroutines to update primal values after state_update().

```
function updateLocalGhostPrim(loc_ghost_holder,
    loc_keys, dist_prim)
    loc_ghost_holder = loc_ghost_holder[:L]
    localkeys = loc_keys[:L]
    for iter in localkeys
        @. loc_ghost_holder[1][iter].prim =
    dist_prim[loc_ghost_holder[1][iter].globalID].
    prim
    end
    return nothing
end
```

Listing 4. Julia function to update primal values after state_update().

Let us now turn to the other main consideration in developing a CFD solver — data communication and synchronization. As an illustration, Listings 3 and 4 present a part of the Fortran and Julia code that update primal values every time after state_update(rk) finishes execution (see Algorithm 1). Fortran uses PETSc [19] while Julia uses DistributedArrays.jl to perform the scatter-gather copies required. In the Fortran solver, all communication calls are handled using PETSc Vector methods which use MPI_Send and MPI_Receive to exchange data. To ensure consistency of results, MPI_Barrier is used to synchronize all MPI processes. In Julia, all subroutines and communication calls are done via @spawn macros to all the processes. These calls are encapsulated with the @sync macro which blocks till all dynamically-enclosed @spawn calls are complete. Although the subroutines are small, the programmer must ensure that they are always executed at the right time and do not cause any race conditions.

The most important thing to note here is that there is no user-written counterpart of this code in the Regent implementation. All data communication and synchronization in Regent is completely hidden from the user and handled automatically by the programming system. Legion, helped by static analysis done by the Regent compiler, dynamically computes exactly which data has to be moved where and inserts the required copies while maintaining program correctness. As an example of implicit data communication and synchronization in Regent, consider the computation of residues. This involves a scalar reduction, shown as part of the Regent code in line 23 in Listing 2. In Fortran and Julia, this is done through an `AllReduce` call. Regent, however, recognizes common patterns that correspond to collectives (such as reductions) and automatically replaces them with the necessary runtime calls to perform the collectives. We should note here that Legion does not use MPI for communication. Rather, Realm [20], the low-level runtime targeted by Legion, is responsible for actual message passing with inter-node communication done using a GASNet [21] networking layer.

One clear advantage of the programming system being completely responsible for data movement is in terms of code readability, writability and maintainability. Programmers no longer need to think about setting up application level synchronization points or spend time debugging race conditions, as is often the case in explicitly parallel programming models like Fortran and Julia.

There are several optimizations available in Regent and Legion for improving scalability and performance of applications. We will briefly discuss those that were instrumental in the current implementation.

- Index launches [7] are an optimization that reduces the runtime analysis cost of a loop of task launches by using static analysis performed by the Regent compiler. All five major task launches in `q-LSKUM` were performed using index launches, as can be seen in Listing 2 by the multiple occurrences of the `__demand(__index_launch)` source code annotation. It should be noted that this annotation does not actually change what the compiler will optimize, but acts as a safeguard by forcing the compiler to throw an error if it is unable to perform the optimization.
- The mapper [1] is a Legion interface to customize the scheduler and manage all performance decisions made by the runtime system, including assigning tasks to cores and data requirements to physical memory. We customized the default mapper to disable automatic load balancing and noticed significantly increased performance on AMD processors because otherwise a subregion might transfer between cores and hurt performance, possibly due to AMD's non-uniform chip inter-connects.
- Dynamic control replication (DCR) [22] is an optimization that improves scalability of Regent and Legion applications by launching a set of long-running worker threads called *shards* on multiple nodes. We used DCR on the top-level task and found it to be crucial for multi-node performance.
- Finally, we used Regent's support for OpenMP code generation in which intra-task parallelism is extracted by converting a for-list or for-num loop to an OpenMP style loop. During testing, we observed that our application was copy-bound for multi-node execution, so we switched to using Regent + OpenMP which greatly improved performance by increasing subregion size and therefore reducing the number of ghost points which have to be copied between nodes. Since the compiler is responsible for all the required code transformations, very little programmer effort was needed to make this switch. On the other hand, in Fortran, a significant amount of time needs to be invested in refactoring an application to use OpenMP. Note that although Regent targets the OpenMP API, it does not use OpenMP source code annotations. All decisions about parallelism are made by the compiler directly and automatically, and the user is only involved in hinting where such optimizations might be beneficial.

In summary, although implicitly parallel programming systems have a lot of clear advantages over more traditional, explicitly parallel approaches, they are not without their limitations. Since Legion has to carry out dynamic dependence analysis of all tasks, having a large number of fine grained tasks will slow down execution by making the runtime analysis the execution bottleneck. We were able to circumnavigate this by inlining a lot of our fine grained tasks, at the cost of some lost parallelism. Moreover, due to how memory instances are currently managed in Realm, we see increased memory usage in Regent compared to our Fortran or Julia implementations.

IV. Results and Discussion

In this section, we present numerical results to demonstrate the performance of the implicitly parallel meshfree `q-LSKUM` solver based on Regent. Its performance is assessed by comparing with explicitly parallel versions of the same code written in Fortran 90 and Julia 1.5.1. Note that, in the present work, Fortran parallel code uses MPI with PETSc libraries, while Julia uses its built-in parallel library. We did not pursue Julia with MPI as we are interested in assessing the performance of the built-in Julia parallel library as an alternative to MPI + X.

Note that the Fortran code is a production code that is being extensively used to compute fluid flows around many aerodynamic configurations. The Regent code is verified on these standard test cases for $2D$ inviscid flows by comparing the output quantities such as lift and drag coefficients, convergence in residue fall and converged flow solution with the values obtained from the Fortran code.

To assess the computational efficiency of the Regent based meshfree solver, we consider the test case of inviscid fluid flow around the NACA 0012 airfoil at Mach number, $M = 0.85$, and angle of attack, $AoA = 1^o$. For the bench-marks, five levels of point distributions are used, ranging from 0.8 to 40 million points. All computations are performed with double precision

TABLE I
CONFIGURATION OF THE COMPUTE NODES.

Node configuration	AMD
Model	AMD EPYC$^{\text{TM}}$ 7542
Cores	64 (32×2)
Core Frequency	2.90 GHz
Global Memory	256 (32×8) GB
Memory Speed	3200 MT/s
$L2$ Cache	32 (16×2) MB
$L3$ Cache	128 (64×2) MB

on compute nodes whose specifications are given in Table I. These nodes are interconnected using a high speed Mellanox EDR InfiniBand network with 100 Gbps fully bi-directional bandwidth.

To measure the performance of the parallel codes, we define a cost metric called the Rate of Data Processing (RDP). The RDP can be defined as the total wall clock time in seconds per iteration per point. It is therefore clear that lower the values of RDP, better the performance of the code. Furthermore, as we increase the size of the point distribution, the RDP should decrease asymptotically. In the present work, the RDP values for all versions of the code are generated by fixing the number of iterations in the flow solver to 1000.

Table II shows a comparison of the RDP values on a single compute node. Here, both Fortran and Julia parallel codes are run by decomposing all five point distributions into 64 partitions. These 64 partitions are executed on 64 cores. On the other hand, Regent uses 62 partitions, while Regent + OpenMP uses 2 partitions. In the case of Regent, 62 partitions are assigned to 62 cores and the remaining 2 cores are used by Legion for runtime dependence analysis and utility processing. As far as Regent + OpenMP is concerned, the 2 partitions are due to the architecture of the AMD node used in the present work. As per the present specification, this node consists of two sockets. Each socket is populated with one physical CPU consisting of 32 cores. The HPC platform is configured in such a way that it consists of one NUMA domain per socket. As NUMA optimizes the spatial availability of the memory, Regent + OpenMP is able to take advantage of it and thus yields better performance. Each partition uses 30 OpenMP threads that are assigned to 30 cores. The remaining 4 cores are used by Legion for operations as described earlier.

TABLE II
COMPARISON OF RDP VALUES ON A SINGLE NODE.

No. of points	Regent	Regent + OpenMP	Fortran	Julia
RDP $\times 10^{-7}$ (Lower is better)				
$804,824$	9.9266	6.8145	4.3367	48.2093
$2,642,264$	4.8180	6.4662	4.0788	31.8098
$9,992,000$	3.7195	6.2460	3.8406	22.2528
$25,330,172$	3.3717	6.6212	3.7374	17.5542
$39,381,464$	2.8772	5.9714	3.6717	15.0160

The tabulated values show that the RDP values decrease continuously with increase in the size of the point distribution. On the first two point distributions, the RDP values based on Regent are observed to be higher than Fortran. This behaviour of Regent can be attributed to Legion's runtime dependence analysis. Note that the complexity of this analysis is independent of the size of the point distribution. On the coarse distributions, the task granularities are smaller, and thus the time spent by Legion to perform this analysis took a significant portion of execution time and therefore resulted in higher RDP values. On the finer distributions, this time factor is found to be negligible compared to the time spent on communications and tasks.

In typical MPI implementations such as Fortran, communications employ `MPI_Barrier` to synchronize the processes. In the present Fortran parallel code, PETSc uses `MPI_Barrier` before computing the `flux_residual()` and after `state_update(rk)`. This ensures that updated values are communicated during `MPI_Send` and `MPI_Recv` calls. Due to this, cores that reach the barrier earlier than others need to wait, which results in CPU idling. On the other hand, in the case of Regent, instead of cores, Legion's dependence analysis works at the level of tasks, their dependencies and data requirements. During mapping, tasks are assigned to cores and their data requirements are assigned to physical memory. Legion will then try to execute these tasks as soon as the assigned cores are free and data requirements are met. Since there are no global synchronization barriers in the present Regent implementation of the solver, all the tasks listed in Listing 2 will only wait for their individual dependencies to be resolved. This causes reduction in CPU idling time and thus results in lower RDP values compared to Fortran on the finer point distributions.

The RDP values based on Regent + OpenMP are observed to be higher compared to pure Regent. Unlike pure Regent, where only inter-task parallelism is extracted, OpenMP code generation parallelizes for-list loops inside tasks. In order to take advantage of this feature, the size of each subregion needs to be increased. However, the number of tasks that are available for analysis reduces as the number of partitions decreases. This results in loss of inter-task parallelism and causes a net performance drop in Regent + OpenMP.

Compared to other parallel codes, the performance of the Julia code is observed to be poor as its RDP values are significantly higher. In the present work, the Julia code employs `DistributedArrays.jl` package as an interface to the Julia parallel framework. During communications within the compute node, instead of the desired data it synchronizes the entire point data. This significantly increases the communication overhead leading to higher RDP values.

To analyse the relative performances of Regent, Regent + OpenMP and Julia codes with respect to the Fortran code on a single node, Figure 1 shows a comparison of the relative RDP. Here, the relative RDP of Regent is defined as the ratio of the RDP values based on the Fortran code to the RDP of the Regent code. Similarly, we can define the relative RDPs of

Fig. 1. Comparison of relative RDP on a single node.

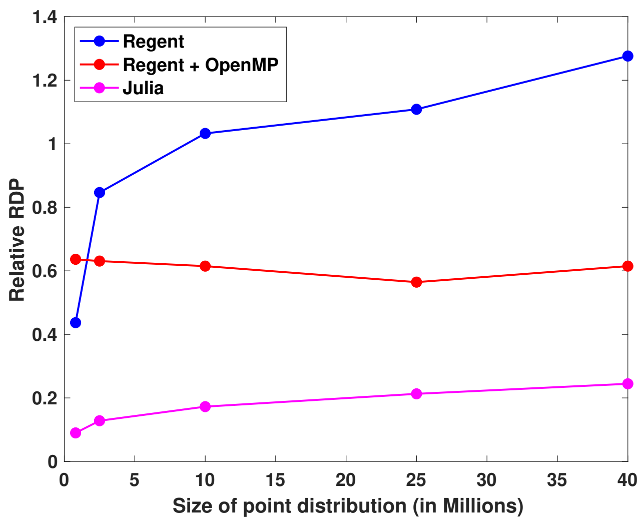
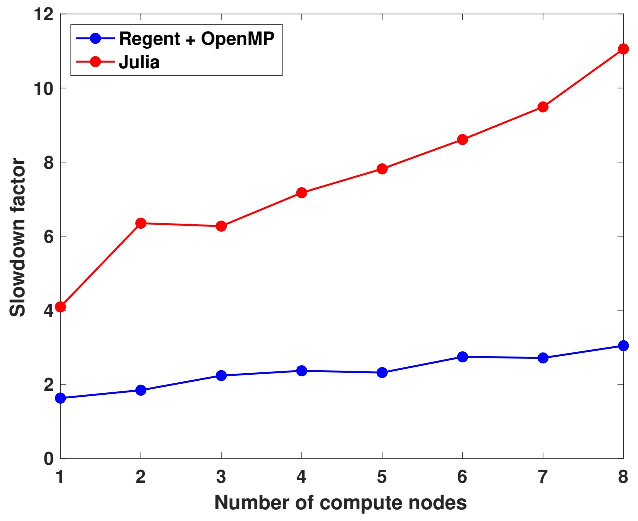


Fig. 2. Comparison of the slowdown factor in RDP values on the finest point distribution.



Regent + OpenMP and Julia. From this plot we can observe that Regent performs better than Fortran on medium size point distributions and exhibits superior performance on the finest point distribution. On the other hand, the relative RDP of Regent + OpenMP is observed to be around 0.6 on all levels of point distribution. For Julia, the relative RDP varied from 0.09 to 0.24.

To test the strong scalability of the parallel codes, Table III shows a comparison of the RDP values for the finest distribution up to 8 compute nodes, which amounts to 512 cores. For the comparisons, we consider Regent + OpenMP as pure Regent code shows poor scalability. This is due to the large number of copies of sparse regions of the ghost points made by Legion.

The tabulated values show that the RDP values of the parallel codes decrease continuously with the increase in the number of nodes. It can be observed that the performance of Regent + OpenMP is still slower than Fortran. Similar to its single node performance, on multiple nodes, Legion's dependence analysis is unable to extract enough inter-task parallelism due to lesser number of partitions per node. This resulted in higher RDP values. On the other hand, Julia parallel code exhibits very poor performance. This is due to the fact that on distributed architecture, Julia's parallel framework uses TCP/IP to connect and transport messages between processes. Therefore, it could not take advantage of high-performance interconnects such as InfiniBand used in the present platform. Another possible reason for its poor performance could be the usage of generic Julia binaries as compilation from source was not possible on the current AMD platform.

Figure 2 shows a comparison of the slowdown factor in RDP values of Regent + OpenMP and Julia codes on the finest point distribution. Here, the slowdown factor of Regent + OpenMP is defined as the ratio of the RDP values based on Regent + OpenMP to the RDP values of the Fortran code. Similarly, we can define the slowdown factor for the Julia code. For the Regent + OpenMP code, the slowdown factor varied from 1.63 on the single node to 3.04 on 8 nodes. On the other hand, the slowdown factor of the Julia code increases continuously with the number of nodes. This behaviour can be attributed to the inter node communications overhead.

Figure 3 shows the strong scalability of the parallel codes on the finest point distribution. From this figure, we can observe that the Fortran code scales almost linearly up to 8 nodes. On the other hand, Regent scales linearly up to two nodes, while Julia scales poorly. This behaviour is expected and consistent with the earlier discussion.

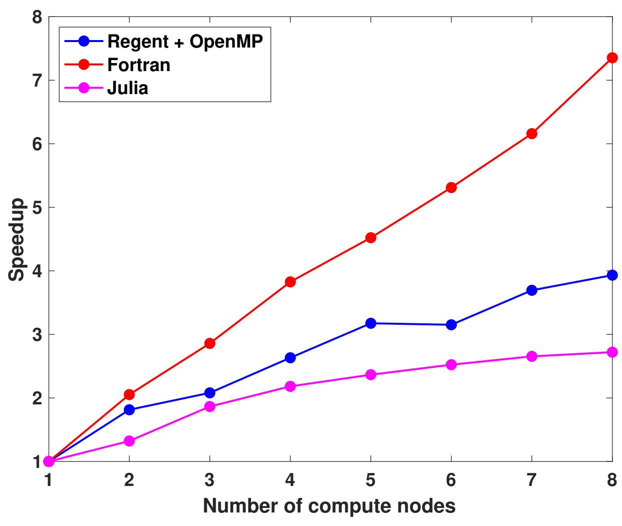
TABLE III
COMPARISON OF RDP VALUES ON MULTIPLE NODES.

Nodes	Regent + OpenMP	Fortran	Julia
RDP values (Lower is better)			
1	5.9714×10^{-7}	3.6717×10^{-7}	1.5016×10^{-6}
2	3.2912×10^{-7}	1.7886×10^{-7}	1.1356×10^{-6}
3	2.8706×10^{-7}	1.2845×10^{-7}	8.0546×10^{-7}
4	2.2686×10^{-7}	9.5952×10^{-8}	6.8814×10^{-7}
5	1.8809×10^{-7}	8.1205×10^{-8}	6.3482×10^{-7}
6	1.8947×10^{-7}	6.9134×10^{-8}	5.9520×10^{-7}
7	1.6165×10^{-7}	5.9616×10^{-8}	5.6575×10^{-7}
8	1.5186×10^{-7}	4.9933×10^{-8}	5.5204×10^{-7}

V. CONCLUSIONS

In this paper, we presented the development of an implicitly parallel meshfree `q-LSKUM` solver based on Regent. The computational efficiency of the Regent solver was assessed by comparing with corresponding explicitly parallel versions of the solver written in Fortran and Julia. To measure the performance of the parallel codes, the RDP values were computed from coarse to very fine distributions.

Fig. 3. Strong scalability on the finest point distribution.



Numerical results for a single node have shown that the performance of the Regent solver was slower compared to the Fortran solver on the coarse distributions. On fine distributions, the RDP values of Regent had surpassed Fortran. Although Regent + OpenMP code performed better than Julia, it was slower than Regent and Fortran on all levels of point distribution. In the case of multi nodes, we were unable to run the Regent solver due to the associated memory issues. The performance of the Regent + OpenMP solver was found to be slower than Fortran but faster than Julia.

As far as the Julia parallel solver's poor performance is concerned, its associated parallel libraries still have memory allocation issues with data communication. Future versions should be able to resolve these issues.

Presently, we are working on enhancing the performance of the Regent solver for distributed execution by optimizing the subregions in order to maximize locality and minimize communications. We are also exploring the feasibility of using the newly added compact sparse instances in Legion and Realm to improve the copy performance and reduce the memory footprint. Having shown very promising results for fine point distributions on a single node, it is worth pursuing Regent for distributed nodes.

Research is also under progress to make the meshfree solver truly hybrid so that it can fully exploit the heterogeneous platforms comprising both CPUs and GPUs. In the future, we would like to extend the present solvers to three-dimensional flows.

Acknowledgments

The authors gratefully acknowledge the computing time provided on the high performance computing facility, Sharanga, at the Birla Institute of Technology and Science - Pilani, Hyderabad Campus. We would like to thank the Legion team for their continuous support. Finally, we thank the anonymous reviewers for their insightful comments and suggestions for improving this paper.

References

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing (SC)*, 2012.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 187–198, Feb. 2011.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[4] T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso, "SU2: An open-source suite for multiphysics simulation and design," *AIAA Journal*, vol. 54, no. 3, pp. 828–846, 2016.

[5] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, "A tensorial approach to computational continuum mechanics using object-oriented techniques," *Computers in Physics*, vol. 12, no. 6, pp. 620–631, 1998.

[6] F. Witherden, A. Farrington, and P. Vincent, "PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, vol. 185, no. 11, pp. 3028–3040, nov 2014. [Online]. Available: https://doi.org/10.1016/j.cpc.2014.07.011

[7] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for HPC with logical regions," in *Supercomputing (SC)*, 2015.

[8] S. M. Deshpande, P. S. Kulkarni, and A. K. Ghosh, "New developments in kinetic schemes," *Computers Math. Applic.*, vol. 35, no. 1, pp. 75–93, 1998.

[9] A. K. Ghosh and S. M. Deshpande, "Least squares kinetic upwind method for inviscid compressible flows," *AIAA paper 1995-1735*, 1995.

[10] J. C. Mandal and S. M. Deshpande, "Kinetic flux vector splitting for Euler equations," *Comp. & Fluids*, vol. 23, no. 2, pp. 447–478, 1994.

[11] S. M. Deshpande, V. Ramesh, K. Malagi, and K. Arora, "Least squares kinetic upwind mesh-free method," *Defence Science Journal*, vol. 60, no. 6, pp. 583–597, 2010.

[12] R. Courant, E. Issacson, and M. Rees, "On the solution of nonlinear hyperbolic differential equations by finite differences," *Comm. Pure Appl. Math.*, vol. 5, pp. 243–255, 1952.

[13] S. M. Deshpande, "On the Maxwellian distribution, symmetric form, and entropy conservation for the Euler equations," *NASA-TP-2583*, 1986.

[14] S. M. Deshpande, K. Anandhanarayanan, C. Praveen, and V. Ramesh, "Theory and application of 3-D LSKUM based on entropy variables," *Int. J. Numer. Meth. Fluids*, vol. 40, pp. 47–62, 2002.

[15] V. Ramesh and S. M. Deshpande, "Least squares kinetic upwind method on moving grids for unsteady Euler computations," *Comp. & Fluids*, vol. 30, no. 5, pp. 621–641, 2001.

[16] J. F. B. M. Kraaijevanger, "Contractivity of Runge-Kutta methods," *BIT Numerical Mathematics*, vol. 31, no. 3, pp. 482–528, 1991.

[17] S. Treichler, M. Bauer, R. Sharma, E. Slaughter, and A. Aiken, "Dependent partitioning," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2016, pp. 344–358.

[18] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[19] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[20] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[21] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *PASCO*, 2007, pp. 24–32.

[22] E. Slaughter, W. Lee, S. Treichler, W. Zhang, M. Bauer, G. Shipman, P. McCormick, and A. Aiken, "Control Replication: Compiling implicit parallelism to efficient SPMD with logical regions," in *Supercomputing (SC)*, 2017.

APPENDIX
ARTIFACT DESCRIPTION

Summary of the Experiments Reported

We performed strong scaling experiments up to 8 nodes. For reproducibility, the exact version of Regent, Fortran and Julia used in the experiments has been saved in a branch, along with all scripts used to build and run.

Artifact Availability

Software Artifact Availability: All author created software artifacts are available in public repositories.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: Some author-created data artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

List of URLs and or DOIs where artifacts are available: Project repositories:

1) For grid generation and sample grids - https://github.com/TestSubjector/QuadTreeMeshSolver
2) For grid partitioning related features - https://github.com/Nischay-Pro/mfpre
3) For Regent specific code - https://github.com/rupanshusoi/meshfree_solver_regent
4) For Fortran specific code - https://github.com/Nischay-Pro/mfcfd
5) For Julia specific code - https://github.com/Nischay-Pro/meshfree-solver

Baseline Experimental Setup, and Modifications Made for the Paper

Relevant Hardware Details: AMD EPYC 7542, InfiniBand EDR Interconnect

Operating Systems and Versions: CentOS 8.2 running Linux kernel 4.18.0-193.6.3

Compilers and Versions: GCC 9.3.0, LLVM 6.0.1 (Regent only), Python 3.8.4 (Legion only), Julia 1.4.2

Libraries and Versions: PETSc 3.13.1, ClusterManagers.jl 0.4.0, DistributedArrays.jl 0.6.5, OpenMPI 3.1.6

Key Algorithms: N/A

Input Datasets and Versions: N/A

Paper Modifications: N/A

Output from scripts that gathers execution environment information:

```
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so
    =01;35:do=01;35:bd=40;33;01:cd=40;33;01:or
    =40;31;01:mi=01;05;37;41:su=37;41:sg=30;43:ca
    =30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.
    tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj
    =01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.
    lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz
    =01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z
    =01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz
    =01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.
    tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz
    =01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.
    rpm=01;31:*.jar=01;31:*.war=01;31:*.ear
    =01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.
    ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z
    =01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.
    swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg
    =01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg
    =01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.
    pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm
    =01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.
    png=01;35:*.svg=01;35:*.svgz=01;35:*.mng
    =01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.
    mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm
    =01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.
    mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv
    =01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.
    rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli
    =01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf
    =01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.
    emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac
    =01;36:*.au=01;36:*.flac=01;36:*.m4a=01;36:*.
    mid=01;36:*.midi=01;36:*.mka=01;36:*.mp3
    =01;36:*.mpc=01;36:*.ogg=01;36:*.ra=01;36:*.
    wav=01;36:*.oga=01;36:*.opus=01;36:*.spx
    =01;36:*.xspf=01;36:
LANG=en_IN.UTF-8
SUDO_GID=1000
HOSTNAME=node1
SUDO_COMMAND=./collect_environment.sh
USER=USER
PWD=/storage/home/nischay/Author-Kit
HOME=/USER
SUDO_USER=nischay
SUDO_UID=1000
MAIL=/var/spool/mail/nischay
SHELL=/bin/bash
TERM=rxvt-unicode
SHLVL=1
LOGNAME=USER
PATH=/sbin:/bin:/usr/sbin:/usr/bin
HISTSIZE=1000
_=/bin/env
Linux node1 4.18.0-193.6.3.el8_2.x86_64 #1 SMP Wed
     Jun 10 11:09:32 UTC 2020 x86_64 x86_64 x86_64
     GNU/Linux
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              64
On-line CPU(s) list: 0-63
Thread(s) per core:  1
Core(s) per socket:  32
Socket(s):           2
NUMA node(s):        2
Vendor ID:           AuthenticAMD
CPU family:          23
Model:               49
Model name:          AMD EPYC 7542 32-Core
    Processor
Stepping:            0
CPU MHz:             2973.915
CPU max MHz:         2900.0000
CPU min MHz:         1500.0000
BogoMIPS:            5799.92
Virtualization:      AMD-V
L1d cache:           32K
L1i cache:           32K
L2 cache:            512K
L3 cache:            16384K
NUMA node0 CPU(s):   0-31
NUMA node1 CPU(s):   32-63
Flags:               fpu vme de pse tsc msr pae
    mce cx8 apic sep mtrr pge mca cmov pat pse36
    clflush mmx fxsr sse sse2 ht syscall nx mmxext
```

```
     fxsr_opt pdpe1gb rdtscp lm constant_tsc
    rep_good nopl nonstop_tsc cpuid extd_apicid
    aperfmperf pni pclmulqdq monitor ssse3 fma
    cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx
    f16c rdrand lahf_lm cmp_legacy svm extapic
    cr8_legacy abm sse4a misalignsse 3dnowprefetch
     osvw ibs skinit wdt tce topoext perfctr_core
    perfctr_nb bpext perfctr_llc mwaitx cpb cat_l3
     cdp_l3 hw_pstate sme ssbd mba sev ibrs ibpb
    stibp vmmcall fsgsbase bmi1 avx2 smep bmi2 cqm
     rdt_a rdseed adx smap clflushopt clwb sha_ni
    xsaveopt xsavec xgetbv1 xsaves cqm_llc
    cqm_occup_llc cqm_mbm_total cqm_mbm_local
    clzero irperf xsaveerptr wbnoinvd arat npt
    lbrv svm_lock nrip_save tsc_scale vmcb_clean
    flushbyasid decodeassists pausefilter
    pfthreshold avic v_vmsave_vmload vgif umip
    rdpid overflow_recov succor smca
MemTotal:       263682004 kB
MemFree:        260224776 kB
MemAvailable:   259382108 kB
Buffers:              240 kB
Cached:            323480 kB
SwapCached:          6336 kB
Active:            214864 kB
Inactive:          133692 kB
Active(anon):       29900 kB
Inactive(anon):     19444 kB
Active(file):      184964 kB
Inactive(file):    114248 kB
Unevictable:            0 kB
Mlocked:                0 kB
SwapTotal:        7815164 kB
SwapFree:         7530108 kB
Dirty:                  0 kB
Writeback:              0 kB
AnonPages:          20800 kB
Mapped:             43768 kB
Shmem:              24480 kB
KReclaimable:      289916 kB
Slab:             1235372 kB
SReclaimable:      289916 kB
SUnreclaim:        945456 kB
KernelStack:        21184 kB
PageTables:          9024 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:    139656164 kB
Committed_AS:     1679312 kB
VmallocTotal:   34359738367 kB
VmallocUsed:            0 kB
VmallocChunk:           0 kB
Percpu:            239616 kB
HardwareCorrupted:      0 kB
AnonHugePages:          0 kB
ShmemHugePages:         0 kB
ShmemPmdMapped:         0 kB
HugePages_Total:        0
HugePages_Free:         0
HugePages_Rsvd:         0
HugePages_Surp:         0
Hugepagesize:        2048 kB
Hugetlb:                0 kB
DirectMap4k:      1703600 kB
DirectMap2M:    132370432 kB
DirectMap1G:    135266304 kB
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda           8:0    1 447.1G  0 disk
   sda1           8:1    1   3.7G  0 part /boot
   sda2           8:2    1 193.7G  0 part
     cl   -root 253:0    0 186.3G  0 lvm  /
     cl   -swap 253:1    0   7.5G  0 lvm  [SWAP]
```

```
[1:0:0:0]    disk    ATA      SAMSUNG MZ7KM480 304
    Q  /dev/sda    480GB
H/W path                Device     Class
    Description
==============================================================

                                   system
    DA700TR-14R4 (To be filled by O.E.M.)
/0                                 bus
    H11DSU-iN
/0/0                               memory
    64KiB BIOS
/0/1e                              memory
    256GiB System Memory
/0/1e/0                            memory
    [empty]
/0/1e/1                            memory
    [empty]
/0/1e/2                            memory
    [empty]
/0/1e/3                            memory
    [empty]
/0/1e/4                            memory
    [empty]
/0/1e/5                            memory
    32GiB DIMM DDR4 Synchronous Registered (
    Buffered) 3200 MHz (0.3 ns)
/0/1e/6                            memory
    [empty]
/0/1e/7                            memory
    32GiB DIMM DDR4 Synchronous Registered (
    Buffered) 3200 MHz (0.3 ns)
/0/1e/8                            memory
    [empty]
/0/1e/9                            memory
    32GiB DIMM DDR4 Synchronous Registered (
    Buffered) 3200 MHz (0.3 ns)
/0/1e/a                            memory
    [empty]
/0/1e/b                            memory
    32GiB DIMM DDR4 Synchronous Registered (
    Buffered) 3200 MHz (0.3 ns)
/0/1e/c                            memory
    [empty]
/0/1e/d                            memory
    [empty]
/0/1e/e                            memory
    [empty]
/0/1e/f                            memory
    [empty]
/0/1e/10                           memory
    [empty]
/0/1e/11                           memory
    32GiB DIMM DDR4 Synchronous Registered (
    Buffered) 3200 MHz (0.3 ns)
/0/1e/12                           memory
    [empty]
/0/1e/13                           memory
    32GiB DIMM DDR4 Synchronous Registered (
    Buffered) 3200 MHz (0.3 ns)
/0/1e/14                           memory
    [empty]
/0/1e/15                           memory
    [empty]
/0/1e/16                           memory
    [empty]
/0/1e/17                           memory
    [empty]
/0/1e/18                           memory
    [empty]
/0/1e/19                           memory
    32GiB DIMM DDR4 Synchronous Registered (
    Buffered) 3200 MHz (0.3 ns)
```

```
/0/1e/1a                          memory
    [empty]
/0/1e/1b                          memory
    32GiB DIMM DDR4 Synchronous Registered (
    Buffered) 3200 MHz (0.3 ns)
/0/1e/1c                          memory
    [empty]
/0/1e/1d                          memory
    [empty]
/0/1e/1e                          memory
    [empty]
/0/1e/1f                          memory
    [empty]
/0/21                             memory
    2MiB L1 cache
/0/22                             memory
    16MiB L2 cache
/0/23                             memory
    128MiB L3 cache
/0/24                             processor
    AMD EPYC 7542 32-Core Processor
/0/49                             memory
    2MiB L1 cache
/0/4a                             memory
    16MiB L2 cache
/0/4b                             memory
    128MiB L3 cache
/0/4c                             processor
    AMD EPYC 7542 32-Core Processor
/0/100                            bridge
    Starship/Matisse Root Complex
/0/100/0.2                        generic
    Starship/Matisse IOMMU
/0/100/7.1                        bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/100/7.1/0                      generic
    Starship/Matisse PCIe Dummy Function
/0/100/7.1/0.2                    generic
    Starship/Matisse PTDMA
/0/100/8.1                        bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/100/8.1/0                      generic
    Starship/Matisse Reserved SPP
/0/100/8.1/0.2                    generic
    Starship/Matisse PTDMA
/0/100/8.1/0.3                    bus
    Starship USB 3.0 Host Controller
/0/100/8.1/0.3/0        usb5      bus
    xHCI Host Controller
/0/100/8.1/0.3/0/2                bus
    Hub
/0/100/8.1/0.3/0/2/1              input
    Keyboard
/0/100/8.1/0.3/1        usb6      bus
    xHCI Host Controller
/0/100/14                         bus
    FCH SMBus Controller
/0/100/14.3                       bridge
    FCH LPC Bridge
/0/101                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/102                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/103                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/104                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/105                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/106                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/107                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/108                            bridge
    Starship Device 24; Function 0
/0/109                            bridge
    Starship Device 24; Function 1
/0/10a                            bridge
    Starship Device 24; Function 2
/0/10b                            bridge
    Starship Device 24; Function 3
/0/10c                            bridge
    Starship Device 24; Function 4
/0/10d                            bridge
    Starship Device 24; Function 5
/0/10e                            bridge
    Starship Device 24; Function 6
/0/10f                            bridge
    Starship Device 24; Function 7
/0/110                            bridge
    Starship Device 24; Function 0
/0/111                            bridge
    Starship Device 24; Function 1
/0/112                            bridge
    Starship Device 24; Function 2
/0/113                            bridge
    Starship Device 24; Function 3
/0/114                            bridge
    Starship Device 24; Function 4
/0/115                            bridge
    Starship Device 24; Function 5
/0/116                            bridge
    Starship Device 24; Function 6
/0/117                            bridge
    Starship Device 24; Function 7
/0/118                            bridge
    Starship/Matisse Root Complex
/0/118/0.2                        generic
    Starship/Matisse IOMMU
/0/118/7.1                        bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/118/7.1/0                      generic
    Starship/Matisse PCIe Dummy Function
/0/118/7.1/0.2                    generic
    Starship/Matisse PTDMA
/0/118/8.1                        bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/118/8.1/0                      generic
    Starship/Matisse Reserved SPP
/0/118/8.1/0.1                    generic
    Starship/Matisse Cryptographic Coprocessor
    PSPCPP
/0/118/8.1/0.2                    generic
    Starship/Matisse PTDMA
/0/118/8.1/0.3                    bus
    Starship USB 3.0 Host Controller
/0/118/8.1/0.3/0        usb7      bus
    xHCI Host Controller
/0/118/8.1/0.3/1        usb8      bus
    xHCI Host Controller
/0/118/8.2                        bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/118/8.2/0                      storage
    FCH SATA Controller [AHCI mode]
/0/118/8.3                        bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/118/8.3/0                      storage
    FCH SATA Controller [AHCI mode]
/0/119                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
```

```
/0/11a                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/11b                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/11c                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/11d                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/11e                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/11f                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/120                           bridge
    Starship/Matisse Root Complex
/0/120/0.2                       generic
    Starship/Matisse IOMMU
/0/120/1.1                       bridge
    Starship/Matisse GPP Bridge
/0/120/1.1/0           enp65s0f0  network
    I350 Gigabit Network Connection
/0/120/1.1/0.1         enp65s0f1  network
    I350 Gigabit Network Connection
/0/120/1.1/0.2         enp65s0f2  network
    I350 Gigabit Network Connection
/0/120/1.1/0.3         enp65s0f3  network
    I350 Gigabit Network Connection
/0/120/7.1                       bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/120/7.1/0                     generic
    Starship/Matisse PCIe Dummy Function
/0/120/7.1/0.2                   generic
    Starship/Matisse PTDMA
/0/120/8.1                       bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/120/8.1/0                     generic
    Starship/Matisse Reserved SPP
/0/120/8.1/0.2                   generic
    Starship/Matisse PTDMA
/0/120/8.2                       bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/120/8.2/0                     storage
    FCH SATA Controller [AHCI mode]
/0/120/8.3                       bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/120/8.3/0           scsi1      storage
    FCH SATA Controller [AHCI mode]
/0/120/8.3/0/0.0.0     /dev/sda   disk
    480GB SAMSUNG MZ7KM480
/0/120/8.3/0/0.0.0/0   /dev/sda   disk
    480GB
/0/120/8.3/0/0.0.0/0/1  /dev/sda1  volume
    3814MiB EXT4 volume
/0/120/8.3/0/0.0.0/0/2  /dev/sda2  volume
    193GiB Linux LVM Physical Volume partition
/0/121                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/122                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/123                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/124                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/125                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/126                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/127                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/128                           bridge
    Starship/Matisse Root Complex
/0/128/0.2                       generic
    Starship/Matisse IOMMU
/0/128/3.1                       bridge
    Starship/Matisse GPP Bridge
/0/128/3.2                       bridge
    Starship/Matisse GPP Bridge
/0/128/3.3                       bridge
    Starship/Matisse GPP Bridge
/0/128/3.3/0                     bus
    ASM1042A USB 3.0 Host Controller
/0/128/3.3/0/0         usb1       bus
    xHCI Host Controller
/0/128/3.3/0/1         usb2       bus
    xHCI Host Controller
/0/128/3.4                       bridge
    Starship/Matisse GPP Bridge
/0/128/3.4/0                     bus
    ASM1042A USB 3.0 Host Controller
/0/128/3.4/0/0         usb3       bus
    xHCI Host Controller
/0/128/3.4/0/1         usb4       bus
    xHCI Host Controller
/0/128/4.1                       bridge
    Starship/Matisse GPP Bridge
/0/128/4.1/0                     bridge
    AST1150 PCI-to-PCI Bridge
/0/128/4.1/0/0                   display
    ASPEED Graphics Family
/0/128/7.1                       bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/128/7.1/0                     generic
    Starship/Matisse PCIe Dummy Function
/0/128/7.1/0.2                   generic
    Starship/Matisse PTDMA
/0/128/8.1                       bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/128/8.1/0                     generic
    Starship/Matisse Reserved SPP
/0/128/8.1/0.2                   generic
    Starship/Matisse PTDMA
/0/129                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/12a                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/12b                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/12c                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/12d                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/12e                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/12f                           bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/130                           bridge
    Starship/Matisse Root Complex
/0/130/0.2                       generic
    Starship/Matisse IOMMU
/0/130/1.1                       bridge
    Starship/Matisse GPP Bridge
/0/130/1.1/0           ib0        network
    MT27800 Family [ConnectX-5]
/0/130/7.1                       bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/130/7.1/0                     generic
    Starship/Matisse PCIe Dummy Function
/0/130/7.1/0.2                   generic
    Starship/Matisse PTDMA
/0/130/8.1                       bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
```

```
/0/130/8.1/0                    generic
    Starship/Matisse Reserved SPP
/0/130/8.1/0.2                  generic
    Starship/Matisse PTDMA
/0/130/8.1/0.3                  bus
    Starship USB 3.0 Host Controller
/0/130/8.1/0.3/0        usb9     bus
    xHCI Host Controller
/0/130/8.1/0.3/1        usb10    bus
    xHCI Host Controller
/0/131                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/132                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/133                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/134                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/135                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/136                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/137                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/138                          bridge
    Starship/Matisse Root Complex
/0/138/0.2                      generic
    Starship/Matisse IOMMU
/0/138/7.1                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/138/7.1/0                    generic
    Starship/Matisse PCIe Dummy Function
/0/138/7.1/0.2                  generic
    Starship/Matisse PTDMA
/0/138/8.1                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/138/8.1/0                    generic
    Starship/Matisse Reserved SPP
/0/138/8.1/0.1                  generic
    Starship/Matisse Cryptographic Coprocessor
    PSPCPP
/0/138/8.1/0.2                  generic
    Starship/Matisse PTDMA
/0/138/8.1/0.3                  bus
    Starship USB 3.0 Host Controller
/0/138/8.1/0.3/0        usb11    bus
    xHCI Host Controller
/0/138/8.1/0.3/1        usb12    bus
    xHCI Host Controller
/0/138/8.2                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/138/8.2/0                    storage
    FCH SATA Controller [AHCI mode]
/0/138/8.3                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/138/8.3/0                    storage
    FCH SATA Controller [AHCI mode]
/0/139                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/13a                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/13b                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/13c                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/13d                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/13e                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/13f                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/140                          bridge
    Starship/Matisse Root Complex
/0/140/0.2                      generic
    Starship/Matisse IOMMU
/0/140/7.1                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/140/7.1/0                    generic
    Starship/Matisse PCIe Dummy Function
/0/140/7.1/0.2                  generic
    Starship/Matisse PTDMA
/0/140/8.1                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/140/8.1/0                    generic
    Starship/Matisse Reserved SPP
/0/140/8.1/0.2                  generic
    Starship/Matisse PTDMA
/0/140/8.2                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/140/8.2/0                    storage
    FCH SATA Controller [AHCI mode]
/0/140/8.3                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/140/8.3/0                    storage
    FCH SATA Controller [AHCI mode]
/0/141                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/142                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/143                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/144                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/145                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/146                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/147                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/148                          bridge
    Starship/Matisse Root Complex
/0/148/0.2                      generic
    Starship/Matisse IOMMU
/0/148/3.1                      bridge
    Starship/Matisse GPP Bridge
/0/148/3.2                      bridge
    Starship/Matisse GPP Bridge
/0/148/7.1                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/148/7.1/0                    generic
    Starship/Matisse PCIe Dummy Function
/0/148/7.1/0.2                  generic
    Starship/Matisse PTDMA
/0/148/8.1                      bridge
    Starship/Matisse Internal PCIe GPP Bridge 0 to
     bus[E:B]
/0/148/8.1/0                    generic
    Starship/Matisse Reserved SPP
/0/148/8.1/0.2                  generic
    Starship/Matisse PTDMA
/0/149                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/14a                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/14b                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/14c                          bridge
    Starship/Matisse PCIe Dummy Host Bridge
```

```
/0/14d                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/14e                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/14f                            bridge
    Starship/Matisse PCIe Dummy Host Bridge
/0/1                              system
    PnP device PNP0c01
/0/2                              system
    PnP device PNP0b00
/0/3                              system
    PnP device PNP0c02
/0/4                              communication
    PnP device PNP0501
/0/5                              communication
    PnP device PNP0501
/0/6                              system
    PnP device PNP0c02
/1                                power
    PWS-1K02A-1R
/2                                power
    PWS-1K02A-1R
```