XPACC: PlasCom2 Development Codelets

Collaborating authors: ***Daniel Bodony, Michael Campbell***

The Center for Exascale Simulation of Plasma-Coupled Combustion
University of Illinois at Urbana-Champaign

The purpose of this document is to describe the codelets used to test the new *PlasComCM*-2.0 design.

1 Codelet Overview & Architecture

The **XPACC** *PlasCom2* development codelets, *Codelet3* and *Codelet4*, are test and pilot implementations of the planned architecture for the upcoming production version of *PlasCom2*, the next-generation successor to the current **XPACC** flagship simulation application, *PlasComCM*. The primary mission for *PlasCom2* development is to provide a platform for physics-based modeling and simulation that is portable and performant on modern (i.e. current and next-generation) high performance computing (HPC) platforms. The development codelets provide a platform for testing the planned architecture, and programming constructs for use in **XPACC**-relevant simulations and environments before committing to final design and implementation details.

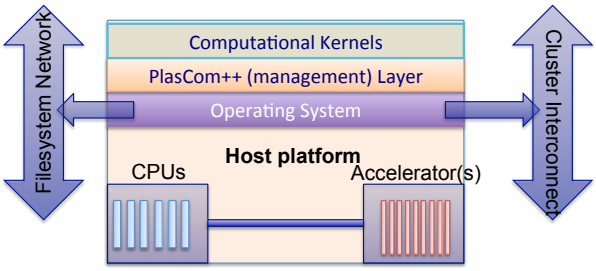

Figure 1: Codelet architecture and targeted platform. The typical modern HPC platform consists of a collection of many heterogeneous nodes, each with some number of multi-core host processors, connected by a thin *data straw* to some form of accelerator device with many integrated processing units. The platform OS provides application interfaces to a internode network, and a parallel file system. The codelets are built up from a C++ management layer called *PlasCom++* (PCPP), which provides the memory management, I/O, MPI, overall control flow, and other aspects of the parallelism, and a *Kernel* layer which provides focused high-performance numerical operations.

It is expected that the current trend of increasing core-counts and decreasing memory-per-core will continue into the coming generations of HPC platforms. The prototypical platform and general high-level software architecture for the codelets are shown in Figure 1. The codelets are built-up from a portable, re-usable C++ management layer called *PlasCom++* (PCPP), and a *Kernel* layer which provides the focused, high-performance numerical operations. The PCPP layer manages the overall control flow of the simulation application, handles the I/O operations, owns and manages the memory, and manages the parallelism through MPI, OpenMP, and accelerator-targeted language constructs. The Kernel layer provides high-performance numerical kernels which have extremely limited logic, and operate only on primitive, built-in data types. Kernel layer constructs are those that are typically transformed, tuned, and targeted for accelerator devices. Kernels can be written in any language that can be interfaced from C++, but are typically written in Fortran.

There are several design objectives to meet for *PlasCom2* among which are:

- Drop-in replacement for *PlasComCM*
- Viable, self-sustainable research tool
- Demonstrate performance on modern platforms
- Address some of the shortcomings of the predecessor codes
  - Performance - I/O, interpolation, general performance on accelerators
  - Overset assembly - needs to be inline, parallel, and dynamic (i.e. support for moving grids)
  - Data structure overhaul - become amenable to CS tool application
  - Repeatability of results
  - Tighter control over development process (w/ testing)

2 Software constructs

2.1 Simulation

2.2 Domain

2.3 Grid

2.4 State

2.5 Advancer

2.6 RHS

2.7 Kernels

3 Codelet Implementation

3.1 Partitions and memory buffers

The following discussion uses integer intervals to describe regions, subregions, and collections (groups) of regions of Cartesian grids, and structured memory buffers. See Appendix A for a brief explanation of intervals the notation used in this document.

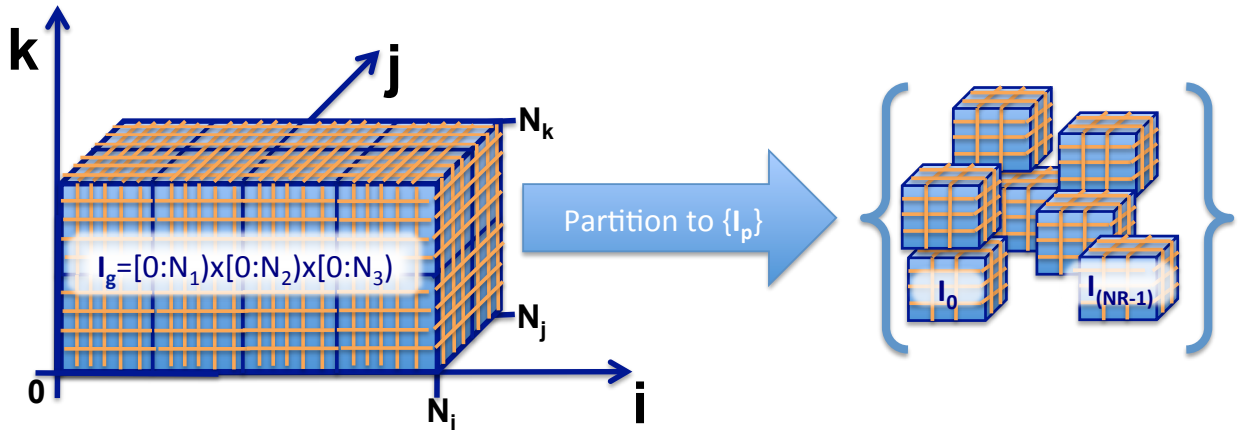

Figure 2: Global domain and partitions.

Given a simulation state with a number NF of fields, each having a number NC_f of components each of datatype width w_f bytes, and a single Cartesian grid as shown in Figure 2 with N_1, N_2, and N_3 points in the computational grid's $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$ directions respectively, and a partitioning or set of NR non-overlapping partitions $\{\mathbf{I}_p\}$ of the overall global grid interval $\mathbf{I}_g$; the memory buffer constructs and requirements to operate a so-called advancer in the codelets is described.

3.1.1 Grid partitions

As shown in Figure 2, the overall global grid interval $\mathbf{I}_g = [0 : N_1) \times [0 : N_2) \times [0 : N_3)$. Each partition in the group of partitions $\{\mathbf{I}_p\}$ (where $0 \leq p < NR$) is handled by a codelet MPI rank. Strictly speaking, a codelet MPI rank may handle more than one partition, however it is useful for this discussion to assume that there is one partition per MPI rank. The concepts and constructs discussed here easily generalize to the case where there is more than one partition per MPI rank. Assuming the typical situation where there one partition per MPI process, it is instructive to think of the index p in $\{\mathbf{I}_p\}$ as denoting the MPI rank of the process that handles a given partition. Each

codelet MPI rank handles a partition with a global interval $\mathbf{I}_p = [s_1 : e_1]_p \times [s_2 : e_2]_p \times [s_3 : e_3]_p$, and $(\mathrm{n}_1, \mathrm{n}_2, \mathrm{n}_3)_p$ grid points in each respective dimension, and total number of points $\mathrm{NP}_p = (\mathrm{n}_1 \cdot \mathrm{n}_2 \cdot \mathrm{n}_3)_p$. In the following discussions, when there is little chance of ambiguity, the p subscript is often dropped when referring to partition-local intervals and quantities. Alternatively, $\hat{p}$, is also used to indicate the selection of a single partition.

3.1.2 Partition halos

If a given partition's interval $\mathbf{I} = [s_1 : e_1] \times [s_2 : e_2] \times [s_3 : e_3]$ does not contain the domain boundary, that is if any of $(s_1, s_2, s_3) \neq (0, 0, 0)$, or any of $(e_1, e_2, e_3) \neq (\mathrm{N}_1 - 1, \mathrm{N}_2 - 1, \mathrm{N}_3 - 1)$, then some so-called *halo* regions will be required. The halo region itself is an interval that represents a subregion of a partition interval that belongs to a remote MPI process. We refer to the halo region representing a non-local interval as the *remote halo* region, and the subregion of the local interval required by neighboring MPI processes as the *local halo* region.

In particular, if any of $s_i \neq 0$, then the partition is said to have a *left* inter-partition boundary in the i direction (for any i up to number of dimensions of the domain). If any of $e_i \neq (\mathrm{N}_i - 1)$, then the partition is said to have a *right* inter-partition boundary. Otherwise, the partition has a domain boundary in the given direction. The number of points in (or width of) the remote halo region required in the i direction is denoted by hw_{i^-} and hw_{i^+} for the left and right halo regions respectively. These widths are determined independently and subject to constraints imposed by the physics, geometry, and numerical methods used in the simulation. A two-dimensional example of this situtation is depicted in Figure 3.

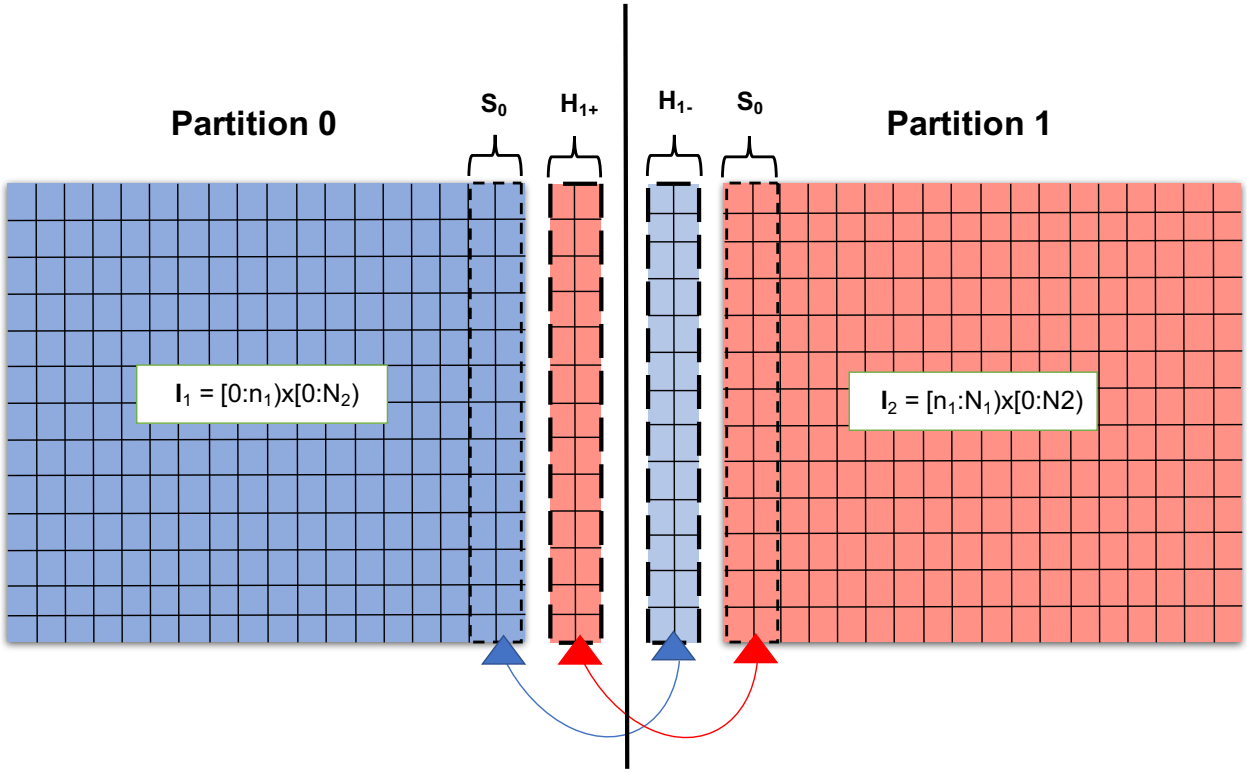

Figure 3: 2D Partition with halo.

Remote halo regions - The collection of required remote halo regions (intervals) $\{\mathbf{H}_{i^\pm}\}$ for a given partition are determined by completely local information. For example, in a 3-dimensional

domain the halo intervals for the $i = 1$ direction are:

$$\mathbf{H}_{1^-} = \begin{cases} [(s_1 - \mathrm{hw}_{1^-}) : s_1) \times [s_2 : e_2] \times [s_3 : e_3], & \text{iff } \; s_1 \neq 0 \\ \emptyset, & \text{otherwise} \end{cases}$$

$$\mathbf{H}_{1^+} = \begin{cases} (e_1 : (e_1 + \mathrm{hw}_{1^+})] \times [s_2 : e_2] \times [s_3 : e_3], & \text{iff } \; e_1 \neq (\mathrm{N}_1 - 1) \\ \emptyset, & \text{otherwise} \end{cases}$$

These extend in the obvious way to form the halo regions in the other directions. The determination of the partition-local $\mathrm{hw}_{i^\pm}$ in each direction $i^\pm$ includes consideration of intra-partition boundaries (e.g. objects in the domain), and the data requirements of the numerical methods used in the simulation. It is important to note that a fully local determination of halo requirements can introduce an asymmetry in the inter-partition halo regions, and thus an imbalance in the communication. These potential imbalances can be addressed by overpartitioning.

Local halo regions - The local halo regions, denoted by $\mathbf{S}_h$, indicate the subregion of the local partition interval $\mathbf{I}$ that contains data needed by remote processors. Given the collection of halo intervals from neighboring processors as $\{\{\mathbf{H}_{i^\pm}\}_h\}$. The local halo regions are found with $\{\mathbf{S}_h\} = \mathbf{I} \cap \{\{\mathbf{H}_{i^\pm}\}_h\} \;\; \forall h \in \{h\}$.

3.1.3 Partition state buffers

Each MPI rank creates a set of NF memory buffers $\{q^f_{ijkc}\}$, one for each simulation state field, corresponding to the partition interval, where $0 \leq f < \mathrm{NF}$, and $0 \leq c < \mathrm{NC}_f$, and $0 \leq (i, j, k) < (\mathrm{n}_1, \mathrm{n}_2, \mathrm{n}_3)$. These are called the *partition state buffers*.

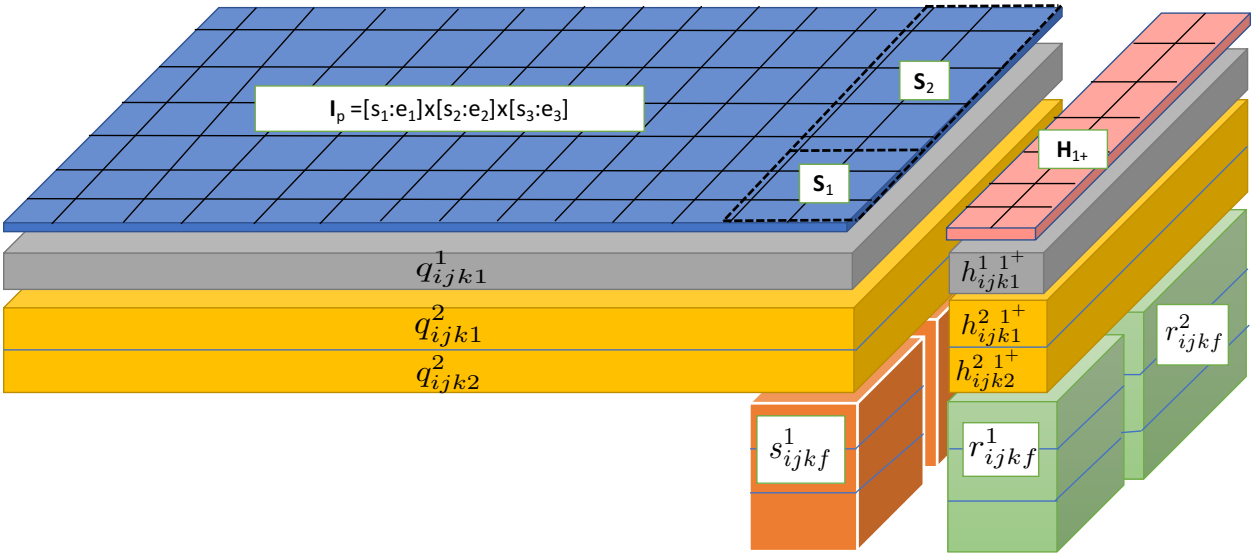

Figure 4: Partition with halo and buffers.

The structure imposed on memory buffers is typically that which corresponds to so-called dictionary ordering wrt i, j, and k. In other words, stride 1 in the $\hat{\mathbf{i}}$ direction, stride n_1 in the $\hat{\mathbf{j}}$ direction, and stride $\mathrm{n}_1 \cdot \mathrm{n}_2$ in the $\hat{\mathbf{k}}$ direction. The ordering of the lower indices in the buffer notation used here is intended to imply the imposed structure, with the inner-most index indicating the stride-1 access. The upper index (f) enumerates the *separate* partition state buffers, i.e. there is a separate partition state buffer for each field. Note that according to this convention, the partition state buffers are arranged to be contiguous for each field, with a continuous block

for each field component. The general setup for a simulation with two fields (one scalar field, and one two-component vector) is illustrated in Figure 4, where the partition state buffer for the scalar field is shown in grey, and the partition state buffer for the two-component vector field is shown in yellow.

The number of bytes required to store a single value of a given field is equal to the number of field components times the byte width of the field datatype ($b_f = w_f \cdot \mathrm{NC}_f$). A memory buffer created to hold a point-bound state field in a particular interval of a Cartesian grid must have a size in bytes equal to the number of points in the interval (buffer) times the number of bytes associated with the field, e.g. the total size of each partition state buffer is $\mathrm{NP}_p \cdot b_f$, for each field f.

3.1.4 Halo and communication buffers

Memory buffers will be required to store and communicate the simulation state field values in the halo region. Additionally, some subregion of the local $\mathbf{I}$ with supporting buffers will be required for communicating the values of the state fields in the local subregion to the remote MPI process.

Remote halo buffers - A set of memory buffers $\{h^{fd^{\pm}}_{ijkc}\}$, one for each non-empty halo region in each direction $d^{\pm}$ is created with the same structure as the partition state field buffer, i.e. contiguous for each field, with continuous component blocks, to store the non-local state field data.

Receive buffers - Having determined the local collection of remote halos $\{\mathbf{H}_{i^{\pm}}\}$, and given the collection of remote partition intervals, $\{\mathbf{I}_p\}$, we can determine the collection of global intervals for the receive regions for a particular partition $\hat{p}$, $\{\mathbf{R}_h\}_{\hat{p}} = \{\mathbf{H}_{i^{\pm}}\} \cap \{\mathbf{I}_p\} \quad \forall p \neq \hat{p}$. The subset of processors (processor ranks) that have non-empty collision intervals with $\{\mathbf{H}_{i^{\pm}}\}$ are referred to as *neighbors*. The collection of all neighbor ranks is called the *neighborhood* and referred to by $\{h\}$.

$\{\mathbf{R}_h\} = [\mathrm{rs}_{h1} : \mathrm{re}_{h1}] \times [\mathrm{rs}_{h2} : \mathrm{re}_{h2}] \times [\mathrm{rs}_{h3} : \mathrm{re}_{h3}]$ represents the global grid intervals on which the state field data values will be received from remote rank h. A set of receive buffers, $\{r^h_{ijkf}\}$, one for each neighbor rank h that will send data the local process, are created for storing the communicated data. Data from the receive buffers is copied into the remote halo buffers for use in computation. Receive buffers are densely packed, i.e. data from all state fields is packed into one buffer, instead of having a separate buffer for each field as is the case for the partition state buffer.

Send buffers - Send buffers are needed to communicate the state field values in the local halo regions $\{\mathbf{S}_h\}$ to the neighboring ranks in $\{h\}$ that need them. The send buffers $\{s^h_{ijkf}\}$ are, like receive buffers, densely packed so that there is one contiguous buffer for all field data, with continuous blocks for each field component. Data are copied from the local halo intervals $\{\mathbf{S}_h\} \subset \mathbf{I}$ to the send buffers and sent to the neighbor ranks.

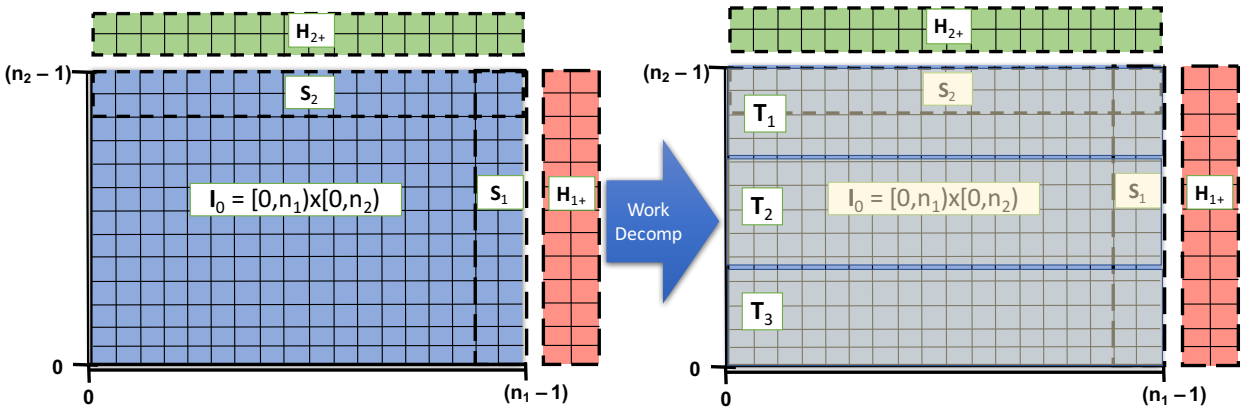

Figure 5: Multi-threaded work decomposition.

3.2 Advancer

The proposed software/code construct to implement stepping (e.g. time marching) in *PlasComCM*-2.0 is the so-called *Advancer*. The implementation of RK4 stepping in a basic single-grid *Advancer* for *Codelet3* and *Codelet4* is described as follows.

4 Examples

4.1 Advect1D

The *advect1d* toy code exercises several of the key software constructs and control flows that are planned for *PlasComCM*-2.0.

4.1.1 Continuous Problem To Be Solved

We solve the one-dimensional advection equation

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, \quad \text{on the domain } \vec{x} \in [0,1]^3,\ t > 0 \tag{1}$$

subject to the initial and boundary conditions

$$u(\vec{x}, 0) = u_0(x), \quad u(0,t) = 0. \tag{2}$$

The general solution is $u(\vec{x}, t) = u_0(x - t)$ where we assume that the initial and boundary data are consistent, $u_0(0) = 0$.

4.1.2 Discrete Problem To Be Solved (1-D Version)

Ignoring for the moment the three-dimensional domain we introduce ideas in 1-D for simplicity. We discretize the x domain $[0,1]$ into N_1 grid points with $\mathrm{N}_1 - 1$ equally spaced intervals of size $\Delta x = 1/(\mathrm{N}_1 - 1)$. We label $\boldsymbol{x} = \{x_i\}_{i=1}^{\mathrm{N}_1}$ the discrete points and $\boldsymbol{u} = \{u_i\}_{i=1}^{\mathrm{N}_1}$ the discrete solution on each of the N_1 points.

We use summation-by-parts (SBP, Strand (1994)) finite difference operators to write the discrete form of $\partial/\partial x$ as $\mathsf{P}^{-1}\mathsf{Q}$ where P and Q have special properties. To apply the boundary condition $u_1(t) = 0$ we use the simultaneous-approximation-term (SAT, Carpenter *et al.* (1994)) approach where an additional term proportional to difference of $u_1(t)$ and the boundary data is added to the right-hand-side of the governing equations.

Putting everything together we get the discrete equation to be solved as

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} + \mathsf{P}^{-1}\mathsf{Q}\boldsymbol{u} = -\tau\mathsf{P}^{-1}\boldsymbol{s}(u_1 - 0), \quad \boldsymbol{u}(0) = \boldsymbol{0}, \tag{3}$$

where $\tau \geq 1/2$ for numerical stability and $\boldsymbol{s} = [1, 0, \ldots, 0]^T$ localizes the boundary condition to the left-most grid point. Looking at the multiplication $\mathsf{P}^{-1}\boldsymbol{s}$ shows that, for diagonal P,

$$\mathsf{P}^{-1}\boldsymbol{s} = (\mathsf{P}_{11})^{-1}\boldsymbol{s}. \tag{4}$$

There are many SBP operators that are possible but the simplest is the second order scheme

$$\mathsf{P} = \Delta x \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 2 & \cdots & 0 & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & 0 & \cdots & 2 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 2 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{bmatrix}, \quad \mathsf{Q} = \begin{bmatrix} -1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 0 & 1 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 0 & \cdots & 0 & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & 0 & \cdots & 0 & -1 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & -1 & 1 \end{bmatrix}. \tag{5}$$

Equation (3) is of the form

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \boldsymbol{R}(\boldsymbol{u}, t) \tag{6}$$

where $\boldsymbol{R}$ is the right-hand-side vector given by

$$\boldsymbol{R}(\boldsymbol{u}, t) = -\mathsf{P}^{-1}\mathsf{Q}\boldsymbol{u} - \tau(\mathsf{P}_{11})^{-1}\boldsymbol{s}(u_1 - 0). \tag{7}$$

We choose the Runge-Kutta 4th order time method to integrate Eq. (6) in time from an initial state $\boldsymbol{u}(0) = \boldsymbol{0}$. The full storage form of the RK4 algorithm is

Data: N, h, $\boldsymbol{u}_0$, CFL
$\boldsymbol{u} \leftarrow \boldsymbol{u}_0$;
$n \leftarrow 1$;
$t \leftarrow 0$;
while $n \leq N$ **do**
  $\boldsymbol{k}_1 \leftarrow h\boldsymbol{R}(\boldsymbol{u}, t)$;
  $\boldsymbol{k}_2 \leftarrow h\boldsymbol{R}(\boldsymbol{u} + (1/2)\boldsymbol{k}_1, t + (1/2)h)$;
  $\boldsymbol{k}_3 \leftarrow h\boldsymbol{R}(\boldsymbol{u} + (1/2)\boldsymbol{k}_2, t + (1/2)h)$;
  $\boldsymbol{k}_4 \leftarrow h\boldsymbol{R}(\boldsymbol{u} + \boldsymbol{k}_3, t + h)$;
  $\boldsymbol{u} \leftarrow \boldsymbol{u} + (1/6)(\boldsymbol{k}_1 + 2(\boldsymbol{k}_2 + \boldsymbol{k}_3) + \boldsymbol{k}_4)$;
  $n \leftarrow n + 1$;
end

4.1.3 Kernels

From the algorithm in the previous section there are several possibilities for kernels that could be used. A simp le one uses these kernels

name	**input**	**output**
init	$\boldsymbol{u}_0$, N, N_1, CFL	$\boldsymbol{u}$, $\boldsymbol{x}$, Δx
rhs	$\boldsymbol{u}$, h, t	$\boldsymbol{k}$
rk4	$\boldsymbol{u}$, $\boldsymbol{k}_1$, $\boldsymbol{k}_2$, $\boldsymbol{k}_3$, $\boldsymbol{k}_4$, n	$\boldsymbol{u}$, n

Another possibility splits the rhs into two parts: the interior and BC. In this case 'rhs_interior' applies $\mathsf{P}^{-1}\mathsf{Q}$ to $\boldsymbol{u}$ while 'rhs_bc' applies $-\tau(\mathsf{P}_{11}^{-1})\boldsymbol{s}$.

A Integer intervals and partitions

To effectively discuss details about software constructs for grid and buffer handling in the codelets, some language, conventions, and notions about integer intervals and their interactions need to be developed.

We consider domains discretized by collections of uniform n-dimensional Cartesian grids with a number of points N_1, N_2, and N_3 in the $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$ directions respectively along with a curvilinear mapping from the Cartesian computational coordinates (χ, ξ, η) to physical coordinates (x, y, z) and its Jacobian $\frac{\partial(\chi,\xi,\eta)}{\partial(x,y,z)}$.

Regions and subregions of the discrete computational domain can be represented by intervals (or pairs of numbers) in the n-dimensional space of positive integers, i.e. $\mathbf{I} \in \mathbb{Z}^{+2n}$. In this document, a notation semi-consistent with programming languages and constructs will be used wherein $[a : b]$ represents the closed interval $[a, a+1, ..., b]$, $[a : b)$ represents the closed interval $[a, a+1, ..., b-1]$, and $(a : b]$ represents the closed interval $[a+1, a+2, ..., b]$, where $a, b \in \mathbb{Z}^+$. A region or interval, $\mathbf{I}$ of the n-dimensional computational space is then described by the Cartesian product of n such intervals, e.g. in 3 dimensions, $\mathbf{I} = \mathbf{I}_1 \times \mathbf{I}_2 \times \mathbf{I}_3$.

An unpartitioned Cartesian grid having N_1, N_2, and N_3 points in the $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$ directions, respectively has the *global* grid interval, $\mathbf{I}_g = [0 : \mathrm{N}_1) \times [0 : \mathrm{N}_2) \times [0 : \mathrm{N}_3)$. A one-dimensional grid interval $\mathbf{I} = [s : e]$ has a number of points $\mathrm{n} = e - s + 1$, and an n-dimensional interval $\mathbf{I} = [s_1 : e_1] \times [s_2 : e_2] \times ... \times [s_n : e_n]$ has a number of points $\mathrm{NP} = \prod_{i=1}^{n}(e_i - s_i + 1)$. An empty or null interval has 0 points and is denoted by $\emptyset$.

A partition or subregion of a 3-dimensional Cartesian grid, for example, is a partition interval $\mathbf{I}_p = [s_1 : e_1] \times [s_2 : e_2] \times [s_3 : e_3] \quad \forall\ s_i \leq e_i < \mathrm{N}_i$. The intersection (or overlap) of two one-dimensional intervals is described as follows.

$$\mathbf{I}_1 \cap \mathbf{I}_2 = \begin{cases} [\max(s_1, s_2), \min(e_1, e_2)], & \text{iff } \ s_2 \geq e_1 \text{ and } s_1 \leq e_2 \\ \emptyset, & \text{otherwise} \end{cases} \tag{8}$$

The intersection of two n-dimensional intervals, $\mathbf{I} = \mathbf{I}_1 \times \mathbf{I}_2 \times ... \times \mathbf{I}_n$ and $\mathbf{J} = \mathbf{J}_1 \times \mathbf{J}_2 \times ... \times \mathbf{J}_n$ is defined as follows.

$$\mathbf{I} \cap \mathbf{J} = \begin{cases} (\mathbf{I}_1 \cap \mathbf{J}_1) \times (\mathbf{I}_2 \cap \mathbf{J}_2) \times ... \times \mathbf{I}_n \cap \mathbf{J}_n, & \text{iff } \ \mathbf{I}_i \cap \mathbf{J}_i \neq \emptyset \ \ \forall\ i \leq n \\ \emptyset, & \text{otherwise} \end{cases} \tag{9}$$

Intervals that do not overlap or intersect are said to be *disjoint*. Sets or collections of intervals are denoted by $\{\mathbf{I}_1, \mathbf{I}_2, ..., \mathbf{I}_m\}$ or simply $\{\mathbf{I}_i\}$. A set of intervals $\{\mathbf{I}_i\}$ is said to be mutually disjoint if $\mathbf{I}_i \cap \mathbf{I}_j = \emptyset \ \ \forall\ i \neq j$. Formally, a *partitioning* of $\mathbf{I}_g$ is a set of non-empty mutually disjoint intervals $\{\mathbf{I}_p\}$, wherein $\cup\{\mathbf{I}_p\} = \mathbf{I}_g$. Informally, we say that our Cartesian grid is partitioned into a set of NR non-overlapping regions (partitions) $\{\mathbf{I}_p\}$ which cover the grid.

B Questions

If $\boldsymbol{R}(\boldsymbol{u},t)$ is a linear mapping, then $\boldsymbol{R}(\boldsymbol{u}+\boldsymbol{v},t) = \boldsymbol{R}(\boldsymbol{u},t) + \boldsymbol{R}(\boldsymbol{v},t)$, and $\boldsymbol{R}(\alpha\boldsymbol{u},t) = \alpha\boldsymbol{R}(\boldsymbol{u},t)$. Expanding out the RK4 algorithm above:

$\boldsymbol{k}_2 \leftarrow h\boldsymbol{R}(\boldsymbol{u} + (1/2)\boldsymbol{k}_1) = h\boldsymbol{R}(\boldsymbol{u}) + (h^2/2)\boldsymbol{R}(\boldsymbol{R}(\boldsymbol{u})) = h\boldsymbol{R}(\boldsymbol{u}) + (h^2/2)\boldsymbol{R}^2(\boldsymbol{u})$

$\boldsymbol{k}_3 \leftarrow h\boldsymbol{R}(\boldsymbol{u} + (1/2)\boldsymbol{k}_2) = h\boldsymbol{R}(\boldsymbol{u}) + (h/2)(\boldsymbol{R}(h\boldsymbol{R}(\boldsymbol{u}) + (h^2/2)\boldsymbol{R}^2(\boldsymbol{u})))$
$= h\boldsymbol{R}(\boldsymbol{u}) + (h^2/2)\boldsymbol{R}^2(\boldsymbol{u}) + (h^3/4)\boldsymbol{R}^3(\boldsymbol{u})$

$\boldsymbol{k}_4 \leftarrow h\boldsymbol{R}(\boldsymbol{u} + \boldsymbol{k}_3) = h\boldsymbol{R}(\boldsymbol{u}) + h\boldsymbol{R}(h\boldsymbol{R}(\boldsymbol{u}) + (h^2/2)\boldsymbol{R}^2(\boldsymbol{u}) + (h^3/4)\boldsymbol{R}^3(\boldsymbol{u})) =$
$= h\boldsymbol{R}(\boldsymbol{u}) + h^2\boldsymbol{R}^2(\boldsymbol{u}) + (h^3/4)\boldsymbol{R}^3(\boldsymbol{u}) + (h^4/4)\boldsymbol{R}^4(\boldsymbol{u})$

$\boldsymbol{u}(t+\Delta t) = \boldsymbol{u}(t) + (1/6)(\boldsymbol{k}_1 + 2(\boldsymbol{k}_2 + \boldsymbol{k}_3) + \boldsymbol{k}_4) =$
$\boldsymbol{u}(t) + h\boldsymbol{R}(\boldsymbol{u}(t)) + (h^2/2)\boldsymbol{R}^2(\boldsymbol{u}(t)) + (h^3/6)\boldsymbol{R}^3(\boldsymbol{u}(t)) + (h^4/24)\boldsymbol{R}^4(\boldsymbol{u}(t))$

This matches the Taylor expansion for a function, $\boldsymbol{f}(t+\Delta t)$ for small Δt; i.e.:

$$\boldsymbol{f}(t+\Delta t) \approx \boldsymbol{f}(t) + \Delta t\frac{d\boldsymbol{f}}{dt} + (\Delta t^2/2)\frac{d^2\boldsymbol{f}}{dt^2} + (\Delta t^3/6)\frac{d^3\boldsymbol{f}}{dt^3} + (\Delta t^4/24)\frac{d^4\boldsymbol{f}}{dt^4} + O(\Delta t^5)$$

References

Carpenter, M. H., Gottlieb, D. & Abarbenel, S. 1994 Time-stable boundary conditions for finite difference schemes involving hyperbolic systems: Methodology and application for high-order compact schemes. *Journal of Computational Physics* **111**, 220–236.

Strand, B. 1994 Summation by parts for finite difference approximations for d/dx. *Journal of Computational Physics* **110**, 47–67.

Acknowledgment

This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

Disclaimer