Rocpart **Developer's Guide**

F.M. Najjar
A. Haselbacher
& S. Balachandar
Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign
2270 Digital Computer Laboratory, MC-258
1304 West Springfield Avenue
Urbana, IL 61801

December 12, 2006

Contents

Chapter 1

Notation

- A_{tile}: tile area
- C: specific heat of particle
- C_m: ratio of critical pool volume to propellant molten volume, usually set to 1
- d_m: particle mean diameter where $d_m = d_{med} \exp(\sigma^2/2)$
- d_{max}: particle maximum diameter
- d_{med}: particle median diameter
- d_{min}: particle minimum diameter
- d_{next}: diameter of the next particle to be ejected
- d_p: particle diameter
- e^{L}: energy of particle
- e^{pool}: total energy in the pool
- $f_{breakup}$: breakup factor
- $\mathbf{f}^{\mathbf{u}}$: drag law
- f^{θ}: thermal drag law
- k^{G}: gas thermal conductivity
- $\dot{m}_b$: local propellant burn rate
- $\dot{m}_{bu}$: mass burn rate
- $\dot{m}_{dep}$: mass deposition rate
- m^{L}: mass of particle
- m^{pool}: mass of constituent c in the pool

- $\dot{m}^{\mathsf{L}}$: mass source term
- N_{next}: superparticle loading of the next particle to be ejected
- Pr: Prandtl number
- Re: Reynolds number
- $t_{burnout}$: burnout time
- T_{tile}: temperature of tile surface
- T^{G}: gas temperature
- T^{L}: particle temperature
- V_a: volume of molten Aluminum
- V_m: mean particle volume, $V_m = \pi d_m^3/6$
- V_{pc}: critical puddle volume
- V_{pool}: pool volume
- v_{nrm}: tile normal velocity
- $\mathbf{u}^{\mathsf{G}}$: gas velocity components
- $\mathbf{u}^{\mathsf{L}}$: particle velocity components
- $\mathbf{x}^{\mathsf{L}}$: particle coordinates
- We: Weber number
- We_c: Critical Weber number
- α: exponent of diameter for burning model
- Δt: timestep size
- $\boldsymbol{\mu}^{\mathsf{L}}$: momentum components of particle
- $\dot{\boldsymbol{\mu}}^{\mathsf{L}}$: momentum source term
- $\dot{e}^{\mathsf{L}}$: energy source term
- $\mu_{\perp}^{pool}$: total normal momentum in the pool
- ϕ^{L}: particle composition
- ϕ_{cq}: mass fraction of each constituent in the propellant
- ϕ^{L}: surface tension of Lagrangian particle
- ψ^{L}: Volume fraction

- $\eta_{collision}$: efficiency of collision
- ρ^{G}: gas density
- μ^{G}: gas kinematic viscosity
- ν^{G}: gas dynamic viscosity
- $\varphi^{\mathbf{u}}$: correction factor for drag law
- φ^{θ}: correction factor for thermal drag law
- σ: standard deviation of particle diameter log-normal distribution
- $\tau^{\mathbf{u}}$: particle response time
- τ^{θ}: particle thermal response time
- χ: mass fraction of individual oxidizer components
- χ_{eff}: effective mass fraction
- ζ: random number uniformly distributed between $[0, 1[$

Further notation can be found in the Rocfluid_MP framework document.

Chapter 2

Introduction

2.1 Goal & Scope

The goals of this developer's guide are two-fold:

1. To enable software developers other than the main developer(s) to modify and extend the Rocpart source code.

2. To enable software developers other than the main developer(s) to compile, install and run the Rocpart source code on new computer systems.

The scope of this developer's guide is all the information required to attain the goals aforementioned above.

2.2 Related Documents

The information contained in this document is supplemented by the following document:

- Rocpart User's Guide.
- Rocflo & Rocflu Developer's Guides.
- Rocflo & Rocflu User's Guides.
- Rocfluid_MP Framework Guide.
- Rocinteract Developer's and User's Guides.

2.3 Purpose & Methods

This developer's guide describes the main features of Rocpart a Lagrangian Particle Tracking Module. This module is developed to study particles motion in a solid propellant rocket motor. The design goal of the module is to implement it with *minimal* modifications to the all-speed compressible fluid solvers, Rocflo & Rocflu.

The present goal focuses on two-phase flow simulation of solid propellant rockets with motion tracking of the large-sized burning particles. A Lagrangian approach has been chosen where individual particles are being tracked over time. The main code characteristics are summarized as follows:

- Stochastic injection model.
- Lagrangian tracking of particle conserved variables, including mass, momentum , energy, and positions.
- Extraction of particle derived variables.
- Full heat release capabilities for particle burning and smoke generation.
- Explicit time integration.
- Multi-way coupling between particle, smoke and gas solvers.
- Flexible expandable framework designed in Fortran-90.
- Highly scalable parallel implementation through block decomposition
  - ▶ MPI used for inter-process communications.
  - ▶ Non-blocking communications used for optimal performance.

Rocpart design has been extended to run in a *serial* and *parallel* modes with Rocflu and several enhanced capabilities have been added, including

- Dynamic memory allocation.
- Initialization module.
- Improved particle tracking algorithm on mixed unstructured meshes for triangular and non-planar quadrilateral faces.
- Particle statistics on surface patches.
- Highly scalable parallel implementation through domain decomposition on mixed unstructured meshes.
  - ▶ Non-blocking communications used for optimal performance.
  - ▶ Dynamically allocated communication buffers.

Further, an Eulerian-based statistics module has been developed to accumulate various time-based variables on the Eulerian mesh.

We would like to acknowledge the contribution of Dr. L. Massa for the pdf-based injection model.

This guide documents the Rocpart version 2.3.4 of 12/07/2005.

Chapter 3

Algorithms & Features

The governing equations for the conversed quantity fields are presented in details in the Rocfluid_MP Framework guide. The reader is referred to the document for a general and extensive discussion. The current implementation of Rocpart allows multiple components forming the Lagrangian particle.

In general, for Rocpart, the evolution of the particle positions, mass, momentum components and energy are written as follows:

$$\frac{d}{dt}\mathbf{x}_q^{\mathsf{L}} = \mathbf{u}_q^{\mathsf{L}}, \tag{3.1}$$

$$\frac{d}{dt}m_{cq}^{\mathsf{L}} = \dot{m}_{cq}^{\mathsf{L}} = \sum_{\alpha} \dot{m}_{\alpha cq}^{\mathsf{L}}, \qquad \text{for } c = 1, 2, \ldots, n^{\mathsf{L}}, \tag{3.2}$$

$$\frac{d}{dt}\boldsymbol{\mu}_q^{\mathsf{L}} = \dot{\boldsymbol{\mu}}_q^{\mathsf{L}} = \sum_{\alpha} \dot{\boldsymbol{\mu}}_{\alpha q}^{\mathsf{L}}, \tag{3.3}$$

$$\frac{d}{dt}e_q^{\mathsf{L}} = \dot{e}_q^{\mathsf{L}} = \sum_{\alpha} \dot{e}_{\alpha q}^{\mathsf{L}}. \tag{3.4}$$

3.1 Injection Stochastic Algorithm

An important component in the Lagrangian representation of aluminum droplets in the simulation of solid rocket motors is an appropriate description of the injection process at the propellant combustion interface (PCI). In the past, several efforts in the literature have simplified the injection process to be at a fixed rate both temporally and spatially. However, the actual injection mechanism as seen in experimental visualizations and measurements is far too complex and is random in both space and time.

The injection process needs to provide during each timestep:

- Number of droplets injected into the core
- Coordinates of the position on the injected surface
- Initial mass, momentum components and energy

We have developed a computational formalism for the random injection of droplets that mimics the natural mechanism occurring in the propellants surface. Visualizations of aluminized propellant

burning show that as burning proceeds, the micron sized aluminum particles included in the propellant melt, agglomerate and form larger molten puddles on the propellant surface. Periodically, either the entire puddle or parts of it are torn off the propellant surface by the hot gas to form droplets of mostly aluminum.

Based on this physical process, associated with each surface panel of the propellant a molten puddle is maintained, which, in general, comprises several material constituents. The pool maintains the following data to represent its state:

- m_c^{pool}: The mass of constituent c in the pool.
- $\mu_\perp^{pool}$: The total normal momentum in the pool.
- e^{pool}: The total energy in the pool.
- t_{pool}: The non-dimensional time until the next particle ejection (can be thought of as a timer that winds down to zero).
- d_{next}: The diameter of the next particle to be ejected.
- N_{next}: The superparticle loading factor of the next particle.

The evolution equations are as follows:

$$\frac{d}{dt}m_c^{pool} = -\dot{m}_b A_{tile}\phi_c, \qquad \text{for } c = 1, 2, \ldots, n^{\mathsf{L}}, \tag{3.5}$$

$$\frac{d}{dt}\mu_\perp^{pool} = -A_{tile}v_{nrm}\sum_c \dot{m}_b\phi_c, \tag{3.6}$$

$$\frac{d}{dt}e^{pool} = -A_{tile}\left(\frac{1}{2}v_{nrm}^2\sum_c\big(\dot{m}_b\phi_c\big) + T_{tile}\sum_c\big(C\dot{m}_b\phi_c\big)\right). \tag{3.7}$$

Two approaches have been implemented in Rocpart. An original formalism described below and an approach, refered to as Conservative Random Ejection Model (CRE) is discussed in an accompanying document.

3.1.1 Original Injection Model

For the original injection model, five steps are required for particle generation: particle selection, pool initialization, pool augmentation, pool depletion, and particle ejection. It is to be noted that the particle selection, pool initialization and pool augmentation are similar for both models. The pool depletion and particle ejection modules differ amongst the original and the CRE approach. We describe briefly each step:

- particle selection: Two models for particle selection have been implemented and are described in Section 3.1.2. The N_{next} is set to a global constant in the current implementation.
- pool initialization: m_c^{pool}, $\mu_\perp^{pool}$, and e^{pool} are set to zero, t_{pool} is set to $-\ln(\xi)$, where ξ is chosen uniformly from the interval $(0, 1)$, and d_{next} and N_{next} are set by particle selection.
- pool augmentation: Given a mass flux of constituent c, with an associated temperature and normal velocity, these are used to augment m_c^{pool}, $\mu_\perp^{pool}$, and e^{pool}.

- pool depletion: This routine begins by setting a non-dimensional time t_{tot} to zero, then ejecting particles (by particle ejection) until $t_{tot} > 1$. Each time that particle ejection indicates a particle is to be ejected, a particle is created by taking the fraction of pool volume in the particle (given by particle ejection) and giving the new particle that fraction of the pool's mass (for each constituent), normal momentum, and energy (decrementing the pool by the same amount, of course). A new "next" particle is then made with particle selection.

- particle ejection: This algorithm is used to determine when the next particle is to be ejected—in particular, whether this occurs during the current time-step. The routine takes an input t_{tot} which is a non-dimensional measure of when the last particle was ejected during this time-step. Whether a particle is ejected depends on what the volume of the pool would be if it were: $v_{new} = v_{pool} - N_{next}v_{next}$. If $v_{new} \leq 0$, a particle cannot be ejected. Otherwise, we use the new pool volume to determine an expected number of particles to be ejected in the current time-step: $E_{eject} = C_m v_m / v_{new}$, where v_m is the mean superparticle volume, and C_m is an input parameter used to control the process. This is used to augment t_{tot}: $t_{tot} = t_{tot} + E_{eject}t_{pool}$. If $t_{tot} \leq 1$, then one can think of the "particle timer" t_{pool} as having wound down to 0, and the particle being ejected (t_{pool} is then reset to $-\ln(\xi)$ for a new ξ, the fraction v_{next}/v_{pool} is returned to pool depletion, and v_{pool} is set to v_{new}). Otherwise, t_{pool} only winds down to some positive number: $t_{pool} = (t_{tot} - 1)/E_{eject}$, to be precise.

3.1.2 Conservative Random Ejection Model

The reader is referred to the documented entitled The Conservative Random Ejection Model accompanying this Developer's Guide. The pseudoFortran implementation of the SCRE process is summarized as follows: There are two input parameters, `beta` and `meanParticleVolume`, and three state variables, `poolVolume`, `countdown`, and `currentParticleVolume`. The main routine takes as input a volume `addedVolume` to be added to the system, updates the state variables, and ejects particles as necessary.

- Initialization:

```
betafac     = 2. * beta * meanParticleVolume**2
ibetafac    = 1. / betafac
poolVolume  = 0.  ! or some other initial value
CALL createNewParticle
```

- Particle creation (`createNewParticle`):

```
xi =                          ! Set to a uniform random number on (0,1)
countdown = -log(xi)
currentParticleVolume =       ! Choose volume of next particle
currentSuperParticleVolume =  ! Multiple volume of next particle
                              ! by superparticle loading
```

- Main routine—augment system by volume `addedVolume`:

```
  remainingVolume = addedVolume  ! initialize remaining volume

creLoop:DO
    poolExcess     = poolVolume - currentSurperParticleVolume
    possibleExcess = poolExcess + remainingVolume

    px2 = 0.
    IF (poolExcess > 0.) px2 = poolExcess**2

    countdownNext = countdown
    IF (possibleExcess > 0.) &
      countdownNext = countdown + ibetafac*(px2 - possibleExcess**2)

    IF (countdownNext > 0.) THEN  ! pool too small to eject particle
      poolVolume = poolVolume  + remainingVolume
      countdown  = countdownNext
      EXIT creLoop

    ELSE                          ! pool big enough to eject particle
      deltaVolume = sqrt(betafac*countdown + px2) - poolExcess
      poolVolume  = poolVolume + deltaVolume - currentSuperParticleVolume
      remainingVolume = remainingVolume - deltaVolume
      CALL ejectParticle
      CALL createNewParticle

    ENDIF
  END DO creLoop
```

3.1.3 Initial and Boundary Conditions at Injection

Once the algorithm has decided that a particle (or superparticle) is to be injected. The Lagrangian particle datastructure (mass, momentum and energy) is filled in based on the tile datastructure. Locations on the tile surface are randomly selected as initial field. Accordingly, the tile datastructure is decreased as the particle is released from the tile surface.

3.1.4 Particle Size Injection Models

The particle size selection models are currently:

- Logarithmic Normal Distribution: Given a median particle diameter d_{med} and a standard deviation σ of $\ln(d_{med})$, the lognormal distribution to determine the diameter of the next particle to be ejected is :

$$Pr(d) = \frac{1}{\sqrt{2\pi}\sigma d} \exp\left(\frac{-1}{2\sigma^2}\left[\ln \frac{d}{d_{med}}\right]^2\right). \tag{3.8}$$

- Skewed Logarithmic Distribution: Given a peak particle diameter d_{peak}, a minimum diameter,d_{min} a maximum particle diameter, d_{max}, and a standard deviation σ of $\ln(d_{med})$, the skewed dis-

tribution is:

$$Pr(d) = \left[1 - \left(\frac{d}{d_{max}}\right)^2\right]\left[1 - \left(\frac{d_{min}}{d}\right)^2\right]\exp\left(\frac{-1}{2\sigma^2}\left[\ln(d) - \alpha\right]^2\right). \tag{3.9}$$

where α is computed as

$$\alpha = ln(d_{peak}) + \frac{2\sigma^2}{\left[(d_{max}/d_{peak})^2 - 1\right]} \tag{3.10}$$

- PDF-based Distribution: Rocpack and Rocpile generate a pdf-based distribution for the particle diameters in specific propellant and the input is provided in a tabulated file. The file name is called `<casename>.plag_injcpdf`.

The steps of the iterative procedure are as follows

- Choose d based on a lognormal distribution with α and σ as a mean and a standard deviation, respectively.
- If d is larger than d_{max}, reject that value.
- Else generate a random number, x between 0 and 1.
- If x is greater than $Pr(d)$ (Eq. 3.9), accept the diameter.
- Else reject and try a new diameter.

3.2 Temporal Discretization

The time discretization of the governing equations is based on a low-storage three-stage Runge-Kutta scheme. The three steps are summarized for a generic variable (ϕ) and rhs (R) to solve the evolution equation, $\frac{d\phi}{dt} = R$, as follows:

$$\phi^{(1)} = \phi^{(n)} - \frac{8}{15}\Delta t \quad R^{(n)}, \tag{3.11}$$

$$\phi^{(2)} = \phi^{(1)} - \frac{5}{12}\Delta t\left[R^{(1)} - \frac{17}{25}R^{(n)}\right], \tag{3.12}$$

$$\phi^{(n+1)} = \phi^{(2)} - \frac{3}{4}\Delta t\left[R^{(2)} - \frac{5}{9}R^{(1)}\right] \tag{3.13}$$

The generic variable (ϕ) represents any of the conserved variables in the mixture carrier phase, aluminum droplets or aluminum oxide smoke particles.

This scheme provides a consistent field as the particle move through the computational domain in particular for parallel simulations. This scheme is also described in the Rocflo & Rocflu Developer's Guides.

3.3 Interaction Models

Rocinteract has been designed to handle various interaction mechanisms to deal with coupled multiphase simulations. The reader is refered to Rocinteract Developer's & Users's Guides for details.

The various models currently implemented are summarized in the subsections below.

3.3.1 Momentum Transfer Models

The momentum transfer between gas and Lagrangian particles is described as follows.

- Response time and Reynolds number:

$$\tau^{\mathbf{u}} = \frac{\rho d_p^2}{18\mu^{\mathsf{G}}} = \frac{m}{3\pi d_p \mu^{\mathsf{G}}}, \qquad \mathrm{Re}^d = \frac{d_p \left|\mathbf{u}^{\mathsf{G}} - \mathbf{u}^{\mathsf{L}}\right|}{\nu^{\mathsf{G}}}. \tag{3.14}$$

- Drag law:

$$\mathbf{f}^{\mathbf{u}} = \varphi^{\mathbf{u}} \frac{\mathbf{u}^{\mathsf{G}} - \mathbf{u}^{\mathsf{L}}}{\tau^{\mathbf{u}}}. \tag{3.15}$$

- Models for $\varphi^{\mathbf{u}}$:
  - ▶ No correction: holds for $\mathrm{Re}^d \lesssim 0.2$

$$\varphi^{\mathbf{u}} = 1 \tag{3.16}$$

  - ▶ Schiller–Naumann correlation: holds for $\mathrm{Re}^d \lesssim 800$

$$\varphi^{\mathbf{u}} = 1 + 0.15 \left(\mathrm{Re}^d\right)^{0.687} \tag{3.17}$$

  - ▶ Clift–Gauvin correlation: holds for $\mathrm{Re}^d \lesssim 3 \times 10^5$

$$\varphi^{\mathbf{u}} = 1 + 0.15 \left(\mathrm{Re}^d\right)^{0.687} + \frac{0.0175}{1 + 4.25 \times 10^4 \left(\mathrm{Re}^d\right)^{-1.16}} \tag{3.18}$$

3.3.2 Thermal Transfer Models

The thermal transfer between gas and Lagrangian particles is described as follows.

- Thermal response time (see (3.14) for Reynolds number)

$$\tau^{\theta} = \frac{C\rho {d_p}^2}{12 k^{\mathsf{G}}} = \frac{H}{2\pi d_p k^{\mathsf{G}}} \tag{3.19}$$

- Thermal drag law

$$f^{\theta} = \varphi^{\theta} \frac{T^{\mathsf{G}} - T^{\mathsf{L}}}{\tau^{\theta}} \tag{3.20}$$

- Models for φ^{θ}:
  - ▶ No correction: holds for $\mathrm{Re}^d \lesssim 0.2$

$$\varphi^{\theta} = 1 \tag{3.21}$$

  - ▶ Ranz–Marshall correlation: holds for $\mathrm{Re}^d \lesssim 5 \times 10^4$

$$\varphi^{\theta} = 1 + 0.3 \left(\mathrm{Re}^d\right)^{1/2} \left(\mathrm{Pr}^{\mathsf{G}}\right)^{1/3} \tag{3.22}$$

3.3.3 Lagrangian Particle Burning Model with Full Heat Release

The combustion of each Al droplet is through a complex process. Once the ambient temperature exceeds the oxide melting point, the oxide shell around the droplet cracks and exposes the liquid Al. Aluminum vaporizes, advects and diffuses away from the droplet, and reacts with the oxidizers (such as O_2, H_2O and CO_2) present. A reaction front forms around the droplet and the primary product of combustion is the Aluminum oxide. Although the reaction thus takes place in the gas phase, here we recognize the fact that on the scale of the rocket we will not be able to resolve the gas phase chemistry that occurs around each droplet. Thus we treat the burning of each droplet in a Lagrangian manner to be associated with each individual droplet. The burn rate of a droplet is assumed to follow a general power-law relation dependent on the droplet diameter, local pressure and temperature, and oxidizer concentration. Here we use the correlation advanced as the model for Al droplet burn rate.

The burn rate for a droplet follows the generalized exponential decay form (kd_p^{α}). With the definition of burn rate, we get:

$$\rho_{Al}\frac{dV_p}{dt} = \rho_{Al}\frac{\pi}{2}d_p^2\frac{d(d_p)}{dt} = -kd_p^{\alpha} \tag{3.23}$$

$$d_p^{2-\alpha}d(d_p) = -\frac{2k}{\pi\rho_{Al}}dt \tag{3.24}$$

Integrating (3.24) from an initial droplet size, d_{p_o}, to burnout, we get the following expression for burnout time, $t_{burnout}$:

$$t_{burnout} = \frac{\pi}{2}\frac{\rho_{Al}}{k}\frac{d_{p_o}^{3-\alpha}}{3-\alpha} \tag{3.25}$$

Beckstead has found a closed formula for $t_{burnout}$ as:

$$t_{burnout,Beckstead} = 1138T^{-1.57}P^{-0.2}d_{p_o}^{1.9}\chi_{eff}^{-0.39}\mathcal{D}_{rel}^{-1} \tag{3.26}$$

In Eq. (3.26), time ($t_{burnout,Beckstead}$) is in ms, temperature (T) is in K, pressure (P) is in atm and d_{p_o} is in μm. Invoking SI Units, Eq. (3.26) converts to:

$$t_{burnout,Beckstead} = 1138T^{-1.57}\big(P/1.101325\times10^5\big)^{-0.2}(d_{p_o}\times10^6)^{1.9}\chi_{eff}^{-0.39}\mathcal{D}_{rel}^{-1} * 10^{-3} \tag{3.27}$$

Equating the two equations we get:

$$\frac{\pi}{2}\frac{\rho_{Al}}{k}\frac{d_{p_o}^{3-\alpha}}{3-\alpha} = 1138T^{-1.57}\big(P/1.101325\times10^5\big)^{-0.2}(d_{p_o}\times10^6)^{1.9}\chi_{eff}^{-0.39}\mathcal{D}_{rel}^{-1} * 10^{-3} \tag{3.28}$$

Hence the exponent d_{p_o} of $3-\alpha = 1.9$, resulting in $\alpha = 1.1$. The constant k in SI units becomes:

$$k = \hat{k}\rho_{Al}T^{1.57}P^{0.2}\chi_{eff}^{0.39}\mathcal{D}_{rel} \tag{3.29}$$

$\hat{k}$ is given by:

$$\hat{k} = \frac{\pi}{2}\frac{1}{(3-\alpha)}\frac{1}{1138}\frac{10^3}{\big(10^6\big)^{1.9}}\big(1.101325\times10^5\big)^{-0.2} \tag{3.30}$$

this results in $\hat{k} = 2.885\times10^{-13}$.

Based on Beckstead's correlation, $\dot{m}_{bu}$, is the burn rate in ($\mathbf{kg/s}$) given as (this equation is based on SI units where d_p is in meters, p_g in Pascals)

$$\dot{m}_{bu} = 2.885 \times 10^{-13} \rho_{Al} T_{\mathsf{L}}^{1.57} P_{\mathsf{L}}^{0.2} \chi_{eff}^{0.39} \mathcal{D}_{rel} d_p^{1.1} \psi^{\mathsf{L}} \tag{3.31}$$

where

$$\chi_{eff} = (\chi_{O_2} + 0.58\chi_{H_2O} + 0.22\chi_{CO_2}) \tag{3.32}$$

$$\mathcal{D}_{rel} = 1.0 + \chi_{H_2}\big(\frac{\mathcal{D}_{oxh}}{\mathcal{D}_{oxhnoh2}} - 1\big) \tag{3.33}$$

with

$$\frac{\mathcal{D}_{oxh2}}{\mathcal{D}_{oxnoh2}} = 3.7 \tag{3.34}$$

and

$$\psi^{\mathsf{L}} = \frac{\phi^{\mathsf{L}} \rho_{AlOx}}{\phi^{\mathsf{L}} \rho_{AlOx} + (1 - \phi^{\mathsf{L}}) \rho_{Al}} \tag{3.35}$$

The above burn rate model was built on the basis of combusion of an individual droplet in a quiescent ambient medium. Conditions under which the *Al* droplets burn within the rocket core differ from this ideal picture in several significant ways. For example, there is strong cross flow between the droplet and the surrounding gas flow in a rocket. Furthermore, interference between the burning of the neighboring droplets may not always be negligible. For lack of better models, we ignore such complexities and use the above burn rate model, as it is the best one available.

The above burn rate model, when naively used, in the context of a solid rocket motor often results in computational difficulty. The gas mixture temperature in localized regions attains unphysical values, far in excess of the boiling point of Al_2O_3, which is typically around 4000^0K. An *adhoc* fix for this problem has been to restrict the heat release from combustion to be only a fraction of the actual value. This of course is not a very satisfactory solution, since it leads to an unphysically lower total heat release to the gas phase. The proper solution for this difficulty lies in the recognition of how the heat is released in the combustion process. The total heat release h of the combustion process is composed of contributions arising from several sub-processes

$$h = -h_{ev} + h_{reac} + h_{cond} + h_{solid}. \tag{3.36}$$

The terms in Eq. (3.36) correspond to the heat of evaporation of Al(l) to Al(g), the heat of reaction from Al(g) to $Al_2O_3(g)$, the heat of condensation from $Al_2O_3(g)$ to $Al_2O_3(l)$, and the heat of solidification from $Al_2O_3(l)$ to $Al_2O_3(s)$. Typical values for these quantities are shown in Table 3.1.

The table clearly shows that the bulk of heat release in the condensation process of $Al_2O_3(g) \rightarrow Al_2O_3(l)$. Localized hotspots appear in the gas phase when the heat of combustion is applied as a whole, without recognizing the different contributions. From physical reasoning it can be argued that provided the local temperature exceeds the boiling point of the oxide, the combustion rection should proceed only as far as $Al(g) \rightarrow Al_2O_3(g)$ and the condensation process cannot locally occur. In such regions, the burning of *Al* droplets must result in the release of only the heat of reaction and the oxide must remain in the gas phase. This points to the need for monitoring Al_2O_3 in the

Table 3.1: Typical values for heats of reaction in Eq. (3.36). Units are J/(kg K).

Quantity	Value
h_{ev}	8240.0
h_{reac}	7157.0
h_{cond}	29326.0
h_{solid}	0.0

gas phase phase, in addition to the smoke particulate phase and the combustion process can be denoted symbolically as

$$2Al(l) + 3O(g) \rightarrow \alpha Al_2O_3(s) + (1-\alpha)Al_2O_3(g)\,, \tag{3.37}$$

where α is the fraction that condenses to solid state as smoke particles. When local temperature (T) is greater than oxide boiling point $T_{b,ox}$, then $\alpha = 0$. When T is just less than $T_{b,ox}$, then α will be determined such that local gas temperature will not exceed $T_{b,ox}$. When $T \ll T_{b,ox}$, then $\alpha = 1$ and furthermore if there is any oxide that still remains in the gas phase it will condense to form the oxide smoke and rease the associated heat of condensation. In the absence of this regulation mechanism, widly varying thermal fields are often generated with very high local temperatures, which in turn can cause spurious numerical instabilities. By incorporating the above physically motivated regulation process here we have a very robust *Al* combustion model.

3.3.4 Scouring Model

To describe the deposition scenario, let us consider the situation where an isolated droplet of diameter, d_p, moving at a relative velocity, $\mathbf{v} - \mathbf{u}_g(\mathbf{x}_p)$, though a cloud of monodispersed smaller smoke particles of diameter, d_s. The number density of the smaller particles (in terms of number of particles per m^3 is given by $6C_{smoke}/\rho_{alox}\pi d_s^3$. Let the larger droplet sweep out a volume over Δt relative to the smaller smoke particles. The relative volume swept out is $\frac{\pi}{4}d_p^2\|\mathbf{u}^{\mathsf{L}} - \mathbf{u}^{\mathsf{G}}(\mathbf{x}_p)\|\Delta t$; this results in the following mass of aluminum oxide within this volume of $\frac{\pi}{4}d_p^2\|\mathbf{u}^{\mathsf{L}} - \mathbf{u}^{\mathsf{G}}(\mathbf{x}_p)\|C_{smoke}\Delta t$

Hence, the deposition rate, $\dot{m}_{dep}$, in (**kg**/**s**) given as

$$\dot{m}_{dep} = \frac{\pi}{4}d_p^2\|\mathbf{u}^{\mathsf{L}} - \mathbf{u}^{\mathsf{G}}(\mathbf{x}_p)\|C_{smoke}\eta_{collision} \tag{3.38}$$

3.4 Breakup Model

A simplified breakup model has been developed in Rocpart and is based on a Weber number defined as:

$$We = \frac{\rho_g d_p(\mathbf{u}^{\mathsf{G}} - \mathbf{u}^{\mathsf{L}})^2}{\phi^{\mathsf{L}}} \tag{3.39}$$

The kernel steps are as follows:

- For each Lagrangian particle, compute the Weber number, We

- Determine the appropriate breakup factor:
  - a constant value based on input
  - a dynamic value computed as:

$$f_{breakup} = \frac{\rho_g d_p (\mathbf{u}^{\mathsf{G}} - \mathbf{u}^{\mathsf{L}})^2}{\phi^{\mathsf{L}} We_c} \tag{3.40}$$

- When the critical condition, $We_c = 10$, is met, the individual particle masses, momenta, energy are decreased by $f_{breakup}$; while the particle loading is increased by $f_{breakup}$.

Chapter 4

Organization

There are three phases in the execution of the Lagrangian particle module, Rocpart, in concert with the all-speed compressible flow solvers, Rocflo and Rocflu:

- Initialization Phase
- Time-stepping Phase
- Solution Output Phase

Slight variations in these phases based on the solver selected are currently presented and are described below:

4.1 Rocflo Phases

4.1.1 Preparation

No automated preparation stage is currently available for Rocpart module with Rocflo solver. The user has to see the datastructure by hand.

4.1.2 Initialization

After the flow solver has initialized its pertinent datastructure, Rocpart module is invoked to setup the datastructure of the Lagrangian particle tracking. This gets activated once the flag `disPartUsed` is set to $.TRUE.$. During this phase, all the arrays are allocated and initialized to zero. Further, the tiles relevant to the injection algorithm are defined and its pertinent datastructure initialized and computed. This stage is initiated in the subroutine (`PLAG_AllocateMemory`).

Further, geometry-based metrics, such as centroids of cell faces and normal vectors of cell faces are computed and saved (`PLAG_RFLO_SetMetrics`). For non-moving grids, the computations are done at the initial step; while for moving grids, they are determined at every timestep.

4.1.3 Timestepping Loop

The main driver kernel for MP simulations is `rungeKuttaMP` routine. At each Runge-Kutta time stepping stage, four distinct steps are taken: (Note that names in parenthesis corresponds to subroutines. All subroutines pertinent to Lagrangian particle have a prefix of `PLAG_`).

- Initialize the overall structure for Rocpart (`PLAG_RkInit`)
  - (i) set the RHS for particle datastructure to zero (`PLAG_ZeroRhs`)
  - (ii) for the initial Runge-Kutta stage, load conserved variables for the particle datastructure from old values (`PLAG_LoadCvOld`)
  - (iii) set the tile RHS to zero (`PLAG_InjcTileZeroRhs`)
  - (iv) for the initial Runge-Kutta stage, load conserved variables for the tile datastructure from old values (`PLAG_InjcTileLoadCvOld`)
- Wrapper to compute all **active** interaction terms (`sourceTermsMP`) through Rocinteract capability
  - (v) compute drag term (`INRT_CalcDrag`)
  - (vi) compute thermal drag term (`INRT_CalcHeatTransferNonBurn`)
  - (vii) compute terms for burning model (`INRT_CalcBurning`)
  - (viii) compute scouring terms (`INRT_CalcScouring`)
- Wrapper routine to evolve the datastructure (`PLAG_RkUpdateWrapper`) which calls two routines, `PLAG_InjcTileUpdate` and `PLAG_Update`. The various components of these routines are described as follows:
  - (ix) compute the tile RHS (`PLAG_InjcTileCalcRhs`)
  - (x) invoke Runge-Kutta update for the tiles (`PLAG_InjcTileRKUpdate`)
  - (xi) compute the particle position vector (`PLAG_CalcRhsPosition`)
  - (xii) invoke Runge-Kutta update (`RkUpdateGeneric`)
  - (xiii) apply the wall bouncing algorithm (`PLAG_WallBounce`)
  - (xiv) invoke the cell particle search algorithm for particle exiting the computational domain (`PLAG_GetCellIndicesOutflow`)
  - (xv) remove exiting particle from infrastructure (`PLAG_PatchRemove DataOutflow`)
  - (xvi) invoke particle location search algorithm (`PLAG_GetCellIndices`)
  - (xvi) at final Runge-Kutta stage, invoke injection algorithm (`PLAG_InjcEjectParticle`)
  - (xvii) invoke breakup algorithm at last RK stage (`PLAG_CalcBreakup`)
- Check status of `PLAG` conserved and derived variables in `AfterUpdateMP`.
  - (xix) check positivity (`PLAG_CheckPositivity`)
  - (xx) check validity (`PLAG_CheckValidity`)
- Wrapper routine to communicate (`PLAG_PatchUpdateWrapper`) part of (`UpdateBoundaryConditionsMP`) (see Section 7 for details).
  - (xxi) communicate data for regions residing on the same processor or for adjacent regions on different processors
  - (xxii) append data from buffers (`PLAG_AppendDatafromBuffers`)
  - (xxiii) obtain derived variable datastructure (`PLAG_CalcDerivedVariables`)

4.1.4 Solution Output

The files pertinent to Rocpart are written once the condition satisfied by `writeTime` is satisfied. Hence,it follows closely the solution write-up of Rocflo (see Rocflo Developer's Guide). This stage is initiated in the subroutine (`PLAG_WriteSolution`). For restart, the file is read from the construct of `PLAG_readSolution`.

The output files generated by Rocpart are created at the same frequency as the solution files created by Rocflo. Several files are created and have the following nomenclature `casename.pdim_`*time* and `casename.plag_sol[a]_`*time*. The first file holds the current number of particles, the maximum number of particles, the number of constituents and the next identifier number. In general, Rocpart saves `aiv,arv,cv` for the `PLAG` datastructure. While for the Tile datastructure, it saves `cv` and `dv(DV_TILE_TIMEFCTR,:)`. If a region has no particles in it, only the number of particles is written. Similar process is undertaken for the tiles. The output files for the Lagrangian particles are written (and read) as follows:

```
time
nDimPlag, nextIdNumber
aivFile
arvFile
cvFile

nDimTile
cvFile
dvFile
```

where `time` is the physical time at which this solution prevails, `nDimPlag` is the number of particles in the computational region, `nextIdNumber` is a counter that keeps track of the particle id's and `nDimTile` is the number of injecting tiles on that computational region.

4.2 Rocflu Phases

4.2.1 Preparation

An automated preparation stage is available for Rocpart module with Rocflu solver. When rfluprep is invoked, the initial particle datastructure is also created based on input flag variables, either based on selected positions or a random distribution. Further details are discussed in the Rocpart User's Guide.

4.2.2 Initialization

After the flow solver has initialized its pertinent datastructure, Rocpart module is invoked to setup the datastructure of the Lagrangian particle tracking. This gets activated once the flag `plagUsed` is set to *.TRUE.*. During this phase, all the arrays are allocated and initialized to zero. Further, the tiles relevant to the injection algorithm are defined and its pertinent datastructure initialized and computed. This stage is initiated in the subroutines (`PLAG_RFLU_AllocMemSol` and `PLAG_RFLU_AllocMemSolTile`).

4.2.3 Timestepping Loop

The main driver kernel for MP simulations is `rungeKuttaMP` routine. At each Runge-Kutta time stepping stage, four distinct steps are taken: (Note that names in parenthesis corresponds to subroutines. All subroutines pertinent to Lagrangian particle running with Rocflu have prefix of `PLAG_` and `PLAG_RFLU_`.

- Initialize the overall structure for Rocpart (`PLAG_RkInit`)
  - (i) set the RHS for particle datastructure to zero (`PLAG_ZeroRhs`)
  - (ii) for the initial Runge-Kutta stage, load conserved variables for the particle datastructure from old values (`PLAG_LoadCvOld`)
  - (iii) set the tile RHS to zero (`PLAG_InjcTileZeroRhs`)
  - (iv) for the initial Runge-Kutta stage, load conserved variables for the tile datastructure from old values (`PLAG_InjcTileLoadCvOld`)
- Wrapper to compute all **active** interaction terms (`sourceTermsMP`) through Rocinteract capability
  - (v) compute drag term (`INRT_CalcDrag`)
  - (vi) compute thermal drag term (`INRT_CalcHeatTransferNonBurn`)
  - (vii) compute terms for burning model (`INRT_CalcBurning`)
  - (viii) compute scouring terms (`INRT_CalcScouring`)
- Wrapper routine to evolve the datastructure (`PLAG_RkUpdateWrapper`) which calls two routines, `PLAG_InjcTileUpdate` and `PLAG_RFLU_Update`. The various components of these routines are described as follows:
  - (ix) compute the tile RHS (`PLAG_RFLU_InjcTileCalcRhs`)
  - (x) invoke Runge-Kutta update for the tiles (`PLAG_InjcTileRKUpdate`)
  - (xi) compute the particle position vector (`PLAG_CalcRhsPosition`)
  - (xii) invoke Runge-Kutta update (`RkUpdateGeneric`)
  - (xiii) invoke particle location search and apply wall bouncing (`PLAG_RFLU_FindCellsTraj`)
  - (xiv) update particle data structure (`PLAG_UpdateDataStruct`)
  - (xvi) at final Runge-Kutta stage, invoke injection algorithm (`PLAG_RFLU_InjectionDriver`)
  - (xvii) invoke breakup algorithm at last RK stage (`PLAG_CalcBreakup`)
  - (xviii) at final Runge-Kutta stage, invoke memory reallocation at last RK stage (`PLAG_ReallocMemWrapper`)
- Check status of `PLAG` conserved and derived variables in `AfterUpdateMP`.
  - (xix) check positivity (`PLAG_CheckPositivity`)
  - (xx) check validity (`PLAG_CheckValidity`)

- Wrapper routine to communicate (PLAG_RFLU_CommDriver) (see Section 7 for details).
  - (xxi) determine number of particles to be communicated in a region across the various borders.
  - (xxii) allocate dynamically memory for send and receive buffers.
  - (xxiii) communicate data for regions residing on the same processor or for adjacent regions on different processors.
  - (xiv) append data from buffers.
  - (xv) deallocate dynamically memory for send and receive buffers.
  - (xvi) keep tracking particles till the trajectory path is complete.
- Wrapper routine to update derived variables (RFLU_SetVarsDiscWrapper)

4.2.4 Solution Output

The files pertinent to Rocpart are written once the condition satisfied by writeTime is satisfied. Hence,it follows closely the solution write-up of Rocflu (see Rocflu Developer's Guide). This stage is initiated in the subroutine (PLAG_RFLU_WriteSolution_ASCII
or PLAG_RFLU_WriteSolution_Binary). For restart, the file is read from the construct of
PLAG_RFLU_ReadSolution_ASCII
or PLAG_RFLU_ReadSolution_Binary.

The output files generated by Rocpart are created at the same frequency as the solution files created by Rocflu. In general, Rocpart saves aiv,arv,cv for the PLAG datastructure. While for the Tile datastructure, it saves cv and dv(DV_TILE_TIMEFCTR,:). If a region has no particles in it, only the number of particles is written. Similar process is undertaken for the tiles. The output file for the Lagrangian particles is written (and read) as follows:

```
# ROCPART dimensions file
# Actual number of particles
       nDimPlag
# Maximum number of particles
    nDimPlagMax
# Number of constituents
       nCont
# Next particle identifier
       nextIdNumber
# End

# ROCFLU particle file
# Precision and range
      15     307
# Physical time
 time
# Dimensions
       nDimPlag      nVars
# Particle x-momentum
   cv(CV_PLAG_XMOM,:)
# Particle y-momentum
```

```
  cv(CV_PLAG_YMOM,:)
# Particle z-momentum
   cv(CV_PLAG_ZMOM,:)
# Particle energy
   cv(CV_PLAG_ENER,:)
# Particle x-location
   cv(CV_PLAG_XPOS,:)
# Particle y-location
   cv(CV_PLAG_YPOS,:)
# Particle z-location
   cv(CV_PLAG_ZPOS,:)
# Particle vapor energy
   cv(CV_PLAG_ENERVAPOR,:)
# Particle mass
   cv(CV_PLAG_LAST+1,:)
# Particle mass
   cv(CV_PLAG_LAST+nCont,:)
# Particle superloading
   arv(ARV_PLAG_SPLOAD,:)
# Particle initial identifier
   aiv(AIV_PLAG_PIDINI,:)
# Particle initial region
   aiv(AIV_PLAG_REGINI,:)
# Particle cell
   aiv(AIV_PLAG_ICELLS,:)
# Patch data

# End
```

where `time` is the physical time at which this solution prevails, `nDimPlag` is the number of particles in the computational region, `nDimPlagMax` is the maximumnumber of particles in the computational region `nextIdNumber` is a counter that keeps track of the particle id's and `nVars` is the number of variables.

Chapter 5

Data Structures

5.1 Philosophy and Abstraction

Rocpart makes heavy use of Fortran 90 user-defined types for the definition of data structures. It is coupled in a modular manner to the Rocfluid_MP datastructure. The basic element of this datastructure is `region`. For each region, a set of variables pertinent to the Lagrangian particle tracking is defined. These include conserved (`cv`), derived (`dv`), transport (`tv`), auxilliary integer (`aiv`), auxilliary real (`arv`), an Eulerian (`ev`) and Eulerian-based statistics (`tav`) variables. Two primary `F90 modules` have been created PLAG_ModParamaters.F90 and ModPartLag.F90.

PLAG_ModParameters defines the generic infrastructure pertinent to Rocpart allowing a modular access to the datastructure. The structure is as follows:

```
MODULE PLAG_ModParameters

  IMPLICIT NONE

! Lagrangian Particles: PLAG ------------------------------------------------------

  INTEGER, PARAMETER :: CV_PLAG_XMOM        = 1, &       ! momentum components
                        CV_PLAG_YMOM        = 2, &
                        CV_PLAG_ZMOM        = 3, &
                        CV_PLAG_ENER        = 4, &
                        CV_PLAG_XPOS        = 5, &       ! position components
                        CV_PLAG_YPOS        = 6, &
                        CV_PLAG_ZPOS        = 7, &
                        CV_PLAG_ENERVAPOR   = 8, &       ! vapor energy
                        CV_PLAG_LAST        = 8          ! final index
                                                         ! see note 2 for index above 9

  INTEGER, PARAMETER :: DV_PLAG_UVEL        = 1,  &      ! particle velocity components
                        DV_PLAG_VVEL        = 2,  &
                        DV_PLAG_WVEL        = 3,  &
                        DV_PLAG_TEMP        = 4,  &      ! static temperature
                        DV_PLAG_DENS        = 5,  &      ! density
                        DV_PLAG_DIAM        = 6,  &      ! diameter
                        DV_PLAG_AREA        = 7,  &      ! cross-sectional area
                        DV_PLAG_VOLU        = 8,  &      ! volume
                        DV_PLAG_SPHT        = 9,  &      ! specific heat
                        DV_PLAG_UVELMIXT    = 10, &      ! mixture velocity components
                        DV_PLAG_VVELMIXT    = 11, &
                        DV_PLAG_WVELMIXT    = 12, &
```

```
                        DV_PLAG_DENSMIXT    = 13, &      ! density
                        DV_PLAG_TEMPMIXT    = 14, &      ! static temperature
                        DV_PLAG_PRESMIXT    = 15, &      ! static pressure
                        DV_PLAG_LAST        = 15         ! final index
                                                         ! see note 2 for index above 16

  INTEGER, PARAMETER :: TV_PLAG_MUELMIXT    = 1, &       ! laminar kinematic viscosity
                                                         ! at particle location
                        TV_PLAG_TCOLMIXT    = 2, &       ! laminar thermal conductivity
                        TV_PLAG_LAST        = 2          ! final index

#ifdef RFLO
  INTEGER, PARAMETER :: AIV_PLAG_PIDINI     = 1, &       ! particle initial id at creation
                        AIV_PLAG_REGINI     = 2, &       ! particle initial region at creation
                        AIV_PLAG_REGCRT     = 3, &       ! particle current region of affiliation
                        AIV_PLAG_ICELLS     = 4, &       ! particle cell index
                        AIV_PLAG_INDEXI     = 5, &       ! particle cell i-index
                        AIV_PLAG_INDEXJ     = 6, &       ! particle cell j-index
                        AIV_PLAG_INDEXK     = 7, &       ! particle cell k-index
                        AIV_PLAG_BURNSTAT   = 8, &       ! particle burning status
                        AIV_PLAG_STATUS     = 9, &       ! particle search status
                        AIV_PLAG_LAST       = 9          ! final index
#endif
#ifdef RFLU
  INTEGER, PARAMETER :: AIV_PLAG_PIDINI     = 1, &       ! particle initial id at creation
                        AIV_PLAG_REGINI     = 2, &       ! particle initial region at creation
                        AIV_PLAG_ICELLS     = 3, &       ! particle cell index
                        AIV_PLAG_BURNSTAT   = 4, &       ! particle burning status
                        AIV_PLAG_STATUS     = 5, &       ! particle search status
                        AIV_PLAG_LAST       = 5          ! final index
#endif

  INTEGER, PARAMETER :: ARV_PLAG_SPLOAD     = 1, &       ! superparticle loading
                        ARV_PLAG_DISTOT     = 2, &       ! total distance travelled
                        ARV_PLAG_LAST       = 2          ! final index

  INTEGER, PARAMETER :: EV_PLAG_DIAM3       = 1,  &      ! eulerian-based particle variables
                        EV_PLAG_DIAM4       = 2,  &
                        EV_PLAG_NUMDENS     = 3,  &      ! number density
                        EV_PLAG_UVEL        = 4,  &      ! velocity components
                        EV_PLAG_VVEL        = 5,  &      !
                        EV_PLAG_WVEL        = 6,  &      !
                        EV_PLAG_TEMP        = 7,  &      !
                        EV_PLAG_MASS        = 8,  &      ! mass
                        EV_PLAG_LAST        = 8          ! final index
                                                         ! see note 3 for index above 8

  INTEGER, PARAMETER :: PLAG_INJC_LOGNORM   = 1, &       ! Injection Models
                        PLAG_INJC_LOGSKWD   = 2, &
                        PLAG_INJC_PDF       = 3

  INTEGER, PARAMETER :: ZEROTH_ORDER        = 0, &       ! Order of accuracy for interpolation
                        FIRST_ORDER         = 1, &
                        SECOND_ORDER        = 2

  INTEGER, PARAMETER :: PLAG_BREAKUP_NOMODEL = 0, &      ! Breakup Models
                        PLAG_BREAKUP_MODEL1  = 1

  INTEGER, PARAMETER :: PLAG_BREAKUP_NOWEBSWI = 0, &     ! Weber Switch for Breakup Model
                        PLAG_BREAKUP_WEBSWI1  = 1

  INTEGER, PARAMETER :: CV_TILE_MOMNRM      = 1, &       ! tile infrastructure
                        CV_TILE_ENER        = 2, &
                        CV_TILE_LAST        = 2          ! final index
```

```
  INTEGER, PARAMETER :: DV_TILE_DIAM        = 1, &       !
                        DV_TILE_SPLOAD      = 2, &
                        DV_TILE_POOLVOLD    = 3, &
                        DV_TILE_COUNTDOWN   = 4, &
                        DV_TILE_LAST        = 4          ! final index

  INTEGER, PARAMETER :: PLAG_STATUS_KEEP    = 0, &       ! Status parameters
                        PLAG_STATUS_COMM    = 1, &
                        PLAG_STATUS_DELETE  = 2, &
                        PLAG_STATUS_LOST    = 3

  INTEGER, PARAMETER :: PLAG_EJEC_MODEL1    = 1, &       ! Ejection Models
                        PLAG_EJEC_CRE       = 2

  INTEGER, PARAMETER :: NPCLS_TOT_MIN     = 1000         ! Minimum Size of Particle DataStructure

  INTEGER, PARAMETER :: FIND_PCL_METHOD_TRAJ_FAST = 0, &
                        FIND_PCL_METHOD_TRAJ_SAFE = 1, &
                        FIND_PCL_METHOD_BRUTE     = 2, &
                        FIND_PCL_METHOD_OCT       = 3, &
                        FIND_PCL_METHOD_LOHNER    = 4

  INTEGER, PARAMETER :: PLAG_SURF_STATS_DIAM3 = 1, &     ! Surface Statistics DataStructure
                        PLAG_SURF_STATS_DIAM4 = 2, &
                        PLAG_SURF_STATS_THETA = 3, &
                        PLAG_SURF_STATS_MOME1 = 4, &
                        PLAG_SURF_STATS_MOME2 = 5, &
                        PLAG_SURF_STATS_MASS  = 6, &
                        PLAG_SURF_STATS_ENER  = 7, &
                        PLAG_SURF_STATS_LAST  = 7

  INTEGER, PARAMETER :: BIN_METHOD_LINEAR  = 1, &        ! Binning Method
                        BIN_METHOD_LOGNORM = 2

END MODULE PLAG_ModParameters
```

ModPartLag defines the datastructure for the input, the Lagrangian particles, the tiles, and the communication infrastructure as follows:

```
MODULE ModPartLag

  USE ModDataTypes
  IMPLICIT NONE

  ! input ------------------------------------------------------------------------

  TYPE t_plag_PDF
     INTEGER              :: nbins,locmax                    ! number of data, maxloc
     REAL(RFREAL)         :: valmax                          ! max val
     REAL(RFREAL),POINTER :: pdfvalues(:,:)
  END TYPE t_plag_PDF

  TYPE t_plag_input
     INTEGER :: nCont                                        ! Total Number of Constituents
     INTEGER :: nPclsTot                                     ! Total Number of Particles in Region
     INTEGER :: ejecModel                                    ! Ejection Model Type
     INTEGER :: injcDiamDist                                 ! Injection Model Type
     INTEGER :: intrplMixtModel                              ! Interpolation Model Type for Mixture
     INTEGER :: nPclsBuffTot                                 ! Total Buffer Size for Patches
     INTEGER :: nPclsBuffCECellsMax                          ! Maximum Buffer Size for Corner and Edge Cells
     INTEGER :: breakupModel                                 ! Breakup Model Type
```

```
   INTEGER :: breakupWebSwi                              ! Weber Switch for Breakup Model
   INTEGER :: readStatus                                 ! Status of reading for input sections
   INTEGER :: nPclsIni                                   ! Number of Initial Particles
   INTEGER :: findPclMethod                              ! Method for finding particles

   INTEGER, POINTER, DIMENSION(:)      :: materialIndex     ! index pointing to appropriate Material

   REAL(RFREAL)                        :: injcVelRatio      ! Injection Velocity Ratio based on Mixture
   REAL(RFREAL)                        :: spLoad            ! Superparticle Loading
   REAL(RFREAL)                        :: injcDiamMean, &   ! Mean Diameter
                                          injcStdDev,   &   ! Standard Deviation
                                          injcDiamMax,  &   ! Maximum Diameter
                                          injcDiamMin       ! Minimum Diameter
   REAL(RFREAL)                        :: injcBeta          ! Beta Coefficient for Ejection
   REAL(RFREAL)                        :: breakupFac        ! Breakup Factor

   REAL(RFREAL)                        :: iniRandDiamMax,  &  ! Initial Maximum Diameter for Random State
                                          iniRandDiamMin,  &  ! Initial Minimum Diameter for Random State
                                          iniRandTempMax,  &  ! Initial Maximum Temperature for Random State
                                          iniRandTempMin,  &  ! Initial Minimum Temperature for Random State
                                          iniRandSpLoadMax,&  ! Initial Maximum Superparticle Loading for Random State
                                          iniRandSpLoadMin    ! Initial Maximum Superparticle Loading for Random State

   REAL(RFREAL), POINTER, DIMENSION(:) :: dens,spht,molw, & ! Input structural parameters
                                          injcMassFluxRatio, &
                                          surftens
   REAL(RFREAL), POINTER, DIMENSION(:) :: iniPosX,   &      ! Initial Xposition
                                          iniPosY,   &      ! Initial Yposition
                                          iniPosZ,   &      ! Initial Zposition
                                          iniDiam,   &      ! Initial Diameter
                                          iniTemp,   &      ! Initial Temperature
                                          iniSpload, &      ! Initial Superparticle Loading
                                          iniVelX,   &      ! Initial Xvelocity
                                          iniVelY,   &      ! Initial Yvelocity
                                          iniVelZ           ! Initial Zvelocity

   CHARACTER(CHRLEN), POINTER, DIMENSION(:) :: materialName ! String represent Material

   TYPE(t_plag_PDF)   :: PDF

END TYPE t_plag_input

! data ----------------------------------------------------------------------

TYPE t_plag
   INTEGER :: nAiv, nArv, nCv, nDv, nEv, nTv
   INTEGER :: nPcls,nPclsPrev,nPclsTot
   INTEGER :: nRequests,nRequestsMetrics
   INTEGER :: nRequestsCECells
   INTEGER :: nRequestsStat
   INTEGER :: nInrtSources
   INTEGER :: nextIdNumber

   INTEGER,      POINTER, DIMENSION(:)   :: requests, requestsMetrics
   INTEGER,      POINTER, DIMENSION(:)   :: requestsI, requestsR
   INTEGER,      POINTER, DIMENSION(:)   :: requestsCECells
   INTEGER,      POINTER, DIMENSION(:)   :: requestsCECellsI,requestsCECellsR
   INTEGER,      POINTER, DIMENSION(:)   :: requestsStat
   INTEGER,      POINTER, DIMENSION(:)   :: cvPlagMass, dvPlagVolu
   INTEGER,      POINTER, DIMENSION(:,:) :: aiv, aivOld

   REAL(RFREAL), POINTER, DIMENSION(:,:) :: arv, arvOld
   REAL(RFREAL), POINTER, DIMENSION(:,:) :: cv,  cvOld, dv, tv
   REAL(RFREAL), POINTER, DIMENSION(:,:) :: rhs, rhsSum
```

```
    REAL(RFREAL), POINTER, DIMENSION(:,:) :: inrtSources

    REAL(RFREAL), POINTER, DIMENSION(:,:,:) :: fc
    REAL(RFREAL), POINTER, DIMENSION(:,:)   :: si, sj, sk
    REAL(RFREAL), POINTER, DIMENSION(:,:)   :: ev, tav
  END TYPE t_plag

  ! tile data structure ------------------------------------------------------

  TYPE t_tile_plag
    INTEGER :: nCv, nDv
    INTEGER,      POINTER, DIMENSION(:)   :: nPclsInjc
    INTEGER,      POINTER, DIMENSION(:)   :: cvTileMass

    REAL(RFREAL), POINTER, DIMENSION(:,:) :: cv, cvOld
    REAL(RFREAL), POINTER, DIMENSION(:,:) :: dv
    REAL(RFREAL), POINTER, DIMENSION(:,:) :: rhs, rhsSum
  END TYPE t_tile_plag

  ! communication data structure ---------------------------------------------

  TYPE t_buffer_plag
    INTEGER :: iRequest
    INTEGER :: nBuffSize
    INTEGER :: nBuffSizeDes
    INTEGER :: nSendBuffI, nRecvBuffI
    INTEGER :: nSendBuffR, nRecvBuffR
    INTEGER :: nBuffSizeTot
    INTEGER :: nSendBuffTotI, nRecvBuffTotI
    INTEGER :: nSendBuffTotR, nRecvBuffTotR
    INTEGER :: iRequestStat
    INTEGER :: nSendBuffStat,nRecvBuffStat
    INTEGER,      POINTER, DIMENSION(:)   :: recvBuffI, sendBuffI
    INTEGER,      POINTER, DIMENSION(:,:) :: aiv, aivOld

    REAL(RFREAL), POINTER, DIMENSION(:)   :: recvBuffR, sendBuffR
    REAL(RFREAL), POINTER, DIMENSION(:,:) :: arv, arvOld, cv, cvOld, dv, tv
    REAL(RFREAL), POINTER, DIMENSION(:,:) :: rhs, rhsSum
    REAL(RFREAL), POINTER, DIMENSION(:)   :: recvBuffStat, sendBuffStat
  END TYPE t_buffer_plag

  ! surface statistics -------------------------------------------------------

  TYPE t_surfstats_plag
    INTEGER, DIMENSION(:), POINTER :: nHits
    REAL(RFREAL), DIMENSION(:,:), POINTER :: vars
  END TYPE t_surfstats_plag


END MODULE ModPartLag
```

5.2 Particle Datastructure

5.2.1 Input Datastructure t_plag_input

These values are read from the input file (`casename.inp`) defined as follows:

- `nCont` corresponds to the total number of constituents that make up the particle. It is an INTEGER with no default value and has to be at least 1.

- nPclsTot represents the maximum number of particles to be evolved in each region. It is an INTEGER with minimum value of 0. If this number is exceeded, the computation crashes. For Rocflu memory reallocation capabilities permit for this number to change during the course of the simulation.
- ejecModel is the ejection model to be invoked. It is an INTEGER currently taking a value of either PLAG_EJEC_MODEL1 or PLAG_EJEC_CRE. The default is set to PLAG_EJEC_MODEL1.
- injcDiamDist is the injection model for the diameter distribution to be invoked. It is an INTEGER currently taking a value of either PLAG_INJC_LOGNORM or PLAG_INJC_LOGSKWD. The default is set to PLAG_INJC_LOGNORM (1).
- intrplMixtModel corresponds to the interpolation model type for the mixture properties. It is an INTEGER taking values of ZEROTH_ORDER, FIRST_ORDER, or SECOND_ORDER. Current support is for ZEROTH_ORDER.
- nPclsBuffTot is the total buffer size for patches used in communication algorithm. It is an INTEGER with minimum value of 0.
- nPclsBuffCECellsMax is the maximum buffer size for corner and edge cells used in communication algorithm. It is an INTEGER with minimum value of 0.
- breakupModel is the breakup model to be invoked. It is an INTEGER currently taking a value of either no model PLAG_BREAKUP_NOMODEL or PLAG_BREAKUP_MODEL1. The default is set to PLAG_BREAKUP_NOMODEL.
- breakupWebSwi is the switch for the breakup factor. It is an INTEGER with the default is set to PLAG_BREAKUP_NOWEBSWI. This allows to dynamically compute the breakup factor breakupFac. It is activated by setting the switch to 1 (or PLAG_BREAKUP_WEBSWI1).
- nPclsIni represents the initial number of particles to be evolved in each region. It is an INTEGER with minimum value of 0.
- readStatus represents the reading status for input sections. It is an INTEGER with initial value of -1 (meaning that the section is unread).
- findPclMethod represents the method for tracking particles in RocfluİIt is an INTEGER currently taking the following values: trajectory-based in a safe mode (FIND_PCL_METHOD_TRAJ_SAFE) and fast mode (FIND_PCL_METHOD_TRAJ_FAST) brute force (FIND_PCL_METHOD_BRUTE), octree (FIND_PCL_METHOD_OCT), known-vicinity (FIND_PCL_METHOD_LOHNER). The default is set to FIND_PCL_METHOD_TRAJ_SAFE. It is recommended not to change this value except for expert users.
- materialIndex represents and index pointing to appropriate material defined in ModMaterials. It is of INTEGER type.
- injcVelRatio corresponds to the injection velocity ratio between the particle and the mixture. It is a REAL with minimum value of 0.0_RFREAL.
- spLoad is the superparticle loading. It is a REAL with a default value of 1.0_RFREAL.

- `injcDiamMean` represents the mean particle diameter at injection. It is a REAL with a default value of $10.0E-6$

- `injcDiamMin` represents the minimum particle diameter at injection. It is a REAL and active for `injcModel` set to `PLAG_INJC_MODEL2`.

- `injcDiamMax` represents the maximum particle diameter at injection. It is a REAL and active for `injcModel` set to `PLAG_INJC_MODEL2`.

- `injcStdDev` is the standard deviation of the particle diameter at injection. It is a REAL with a minimum value of 0.0_RFREAL.

- `injcBeta` corresponds to the beta coefficient for ejection. It is a REAL with a default value of 1.0_RFREAL.

- `iniRandDiamMax,iniRandDiamMin` are the initial maximum and minimum diameters for the random state. It is a REAL with the default is set to `REAL(CRAZY_VALUE_INT)`.

- `iniRandTempMax,iniRandTempMin` are the initial maximum and minimum temperatures for the random state. It is a REAL with the default is set to `REAL(CRAZY_VALUE_INT)`.

- `iniRandSpLoadMax,iniRandSploadMin` are the initial maximum and minimum superparticle loading for the random state. It is a REAL with the default is set to `REAL(CRAZY_VALUE_INT)`.

- `dens` represents the density of each constituent. It is a REAL array with minimum dimension 1.

- `spht` is the specific heat of each constituent. It is a REAL array with minimum dimension 1.

- `molw` corresponds to the constituent molecular weight. It is a REAL array with minimum dimension 1.

- `injcMassFluxRatio` is the mass flux ratio of each constituent.It is a REAL array with minimum dimension 1.

- `surftens` corresponds to the constituent surface tension. It is a REAL array with minimum dimension 1.

- `iniPosX,iniPosY,iniPosZ` represent the initial positions in the x, y, z directions. They are REAL arrays with minimum dimension 1.

- `iniDiam,iniTemp,iniSpLoad` represent the initial diameter, temperature and superparticle loading. They are REAL arrays with minimum dimension 1.

- `iniVelX,iniVelY,iniVelZ` represent the initial velocity in the x, y, and z-directions. They are REAL arrays with minimum dimension 1.

5.2.2 PLAG Datastructure t_plag

The variables in the t_plag are defined as follows:

- nAiv is the first dimension size of the integer auxilliary variable array (aiv). It is of INTEGER type with value AIV_PLAG_LAST.
- nArv is the first dimension size of the real auxilliary variable array (aiv). It is of INTEGER type with value ARV_PLAG_LAST.
- nCv is the first dimension size of the conserved variable array (cv). It is of INTEGER type with value CV_PLAG_LAST+nCont.
- nDv is the first dimension size of the derived variable array (dv). It is of INTEGER type with value DV_PLAG_LAST+nCont.
- nEv is the first dimension size of the Eulerian-based variable array (ev). It is of INTEGER type with value EV_PLAG_LAST+nCont.
- nTv is the first dimension size of the derived variable array (tv). It is of INTEGER type with value TV_PLAG_LAST.
- nPcls is the current number of particles in the region. It is of INTEGER type.
- nPclsTot is the total number of particles in the region used for the particle id. It is of INTEGER type.
- nInrtSources is the total number of interactions. It is of INTEGER type.
- nextIdNumber is a counter for the next particle Id number to use. It is of INTEGER type.
- cvPlagMass is a one dimensional array of size nCont holding the extent for the conversed variable array (cv). It is of INTEGER type computed for each constituent as follows (cvPlagMass(iCont) = CV_PLAG_LAST + iCont).
- dvPlagVol is a one dimensional array of size nCont holding the extent for the derived variable array (dv). It is of INTEGER type computed for each constituent as follows (dvPlagVol(iCont) = DV_PLAG_LAST + iCont).
- aiv and aivOld are two-dimensional arrays of size (nAiv,nPclsTot) for the integer auxilliary variable array (aiv). It is of INTEGER type invoked as aiv(AIV_PLAG_PIDINI,iPcl).
- disPartBurning is a one-dimensional array of size nPclsTot of type LOGICAL set to .TRUE. when the particle is burning.
- arv and arvOld are two-dimensional array of size (nArv,nPclsTot) for the real auxilliary variable array (arv). They are of REAL type invoked as arv(ARV_PLAG_SPLOAD,iPcl).
- cv and cvOld are two-dimensional arrays of size (nCv,nPclsTot) for the conserved variable array (cv). They are of REAL type invoked as cv(CV_PLAG_XMOM,iPcl).

- dv is a two-dimensional array of size (nDv,nPclsTot) for the derived variable array (dv). It is of REAL type invoked as dv(DV_PLAG_UVELMIXT,iPcl).
- tv is a two-dimensional array of size (nTv,nPclsTot) for the transport variable array (tv). It is of REAL type invoked as dv(TV_PLAG_MUELMIXT,iPcl).
- rhs and rhsSum are two-dimensional arrays of size (nCv,nPclsTot) for the RHS of the conserved variable array (cv). They are of REAL type invoked as rhs(CV_PLAG_XMOM,iPcl).
- inrtSources is a two-dimensional array of size (maxEdges,nPclsTot) to compute the source terms in RocinteractIt is of REAL type invoked as inrtSources(INRT_DRAG_L_XMOM_G,iPcl).
- fc is a three-dimensional array of size (ZCOORD,KCOORD,nNodesTot) that holds the face centroids. It is of REAL type and pertinent for Rocflo.
- si, sj, and sk are two-dimensional arrays of size (ZCOORD,nNodesTot) for the face normal vectors. They are of REAL type and pertinent for Rocflo.
- ev is a two-dimensional array of size (nEv,ibc:iec) for the Eulerian-based variable array (ev). It is of REAL type invoked as ev(EV_PLAG_NUMDENS,ic).
- tav is a two-dimensional array of size (plagNStat,ibc:iec) for the Eulerian-based statistics variable array (tav). It is of REAL type invoked as tav(iVar,ic).

5.2.3 Tile Datastructure t_tile_plag

The variables in the t_tile_plag are activate for injection-based boundary conditions for patches and defined as follows:

- nCv corresponds to the first dimension size of the tile conserved variable array (cv). It is of INTEGER type with value CV_TILE_LAST+nCont.
- nDv is the first dimension size of the derived variable array (dv). It is of INTEGER type with value DV_TILE_LAST.
- nPclsInjc represents the current number of particles injected in each tile. It is a dimensional array of INTEGER type and size nTile.
- cvTileMass is a one dimensional array of size nCont holding the extent for the conversed variable array (cv). It is of INTEGER type computed for each constituent as follows (cvPTileMass(iCont) = CV_Tile_LAST + iCont).
- cv and cvOld are two-dimensional arrays of size (nCv,nTile) for the conserved variable array (cv). They are of REAL type invoked as cv(CV_TILE_ENER,iTile).
- dv is a two-dimensional array of size (nDv,nTile) for the derived variable array (dv). It is of REAL type invoked as dv(DV_PLAG_TIMEFCTR,iTile).
- rhs and rhsSum are two-dimensional arrays of size (nCv,nTile) for the RHS of the conserved variable array (cv). They are of REAL type invoked as rhs(CV_TILE_ENER,iTile).

5.2.4 PLAG buffer datastructure t_buffer_plag

(Rocflo Only)

The buffering procedure for multi-processing simulations in Rocpart follows that devised for Rocflo. Details are provided in the Rocflo Developer's Guide. The variables in the t_buffer_plag are activate for patches on communicating regions and defined as follows:

- nBuffSize corresponds to buffer size for the communication arrays. It is of INTEGER type with value varying depending on the number of particles exiting the region.
- nSendBuffI, nRecvBuffI are the send and receive buffer sizes for the Integer-based arrays,(aiv,aivOld). It is of INTEGER type.
- nSendBuffR, nRecvBuffR are the send and receive buffer sizes for the Real-based arrays, (arv,arvOld,cv,cvOld,dv,tv,rhs,rhsSum). It is of INTEGER type.
- nSendBuffR, nRecvBuffR are the send and receive buffer sizes for the Real-based arrays, (arv,arvOld,cv,cvOld,dv,tv,rhs,rhsSum). It is of INTEGER type.
- nSendBuffL, nRecvBuffL are the send and receive buffer sizes for the Logical-based arrays,(distPartBurning). It is of INTEGER type.
- nBuffSizeTot,nSendBuffTotI, nRecvBuffTotI, nSendBuffTotR, nRecvBuffTotR,nSendBuffTotL, nRecvBuffTotL, represent the corresponding variables for the total size of the arrays allocated. This permits some additional storage for the communication arrays.
- aiv and aivOld are two-dimensional arrays of size (nAiv,nBuffSizeTot) for the integer auxilliary variable array (aiv). It is of INTEGER type invoked as aiv(AIV_PLAG_PIDINI,iBuff).
- recBuffI and sendBuffI are one-dimensional arrays of size (nSendBuffTotI) which load all the integer auxilliary variable array (aiv). It is of INTEGER type.
- recBuffR and sendBuffR are one-dimensional arrays of size (nSendBuffTotR) which load all the real variable array (arv,arvOld,cv,cvOld,dv,tv,rhs,rhsSum). It is of REAL type.
- arv and arvOld are two-dimensional arrays of size (nArv,nBuffSizeTot) for the integer auxilliary variable array (arv). It is of REAL type invoked as arv(ARV_PLAG_SPLOAD,iBuff).
- cv and cvOld are two-dimensional arrays of size (nCv,nBuffSizeTot) for the conserved variable array (cv). They are of REAL type invoked as cv(CV_PLAG_XPOS,iBuff).
- dv is a two-dimensional array of size (nDv,nBuffSizeTot) for the derived variable array (dv). It is of REAL type invoked as dv(DV_PLAG_TIMEFCTR,iBuff).
- rhs and rhsSum are two-dimensional arrays of size (nCv,nBuffSizeTot) for the RHS of the conserved variable array (cv). They are of REAL type invoked as rhs(CV_PLAG_YMOM,iBuff).

5.2.5 PLAG Surface Statistics datastructure t_surfstats_plag (Rocflu Only)

The surface statistics procedure for multiple patches in Rocpart follows that devised for Rocflu. Details are provided in the Rocflu Developer's Guide. The variables in the t_surfstats_plag are activate for patches and defined as follows:

- pPatch%surfPlag is a one-dimensional pointer of size pPatch%nBFaces and holds the infrastructure for the surface patch statistics. It is of POINTER type.
- nHits is a one-dimensional array of size nBins, currently set to 20 and represents the number of hits for each bin on the surface patch. It is of INTEGER type.
- vars is two-dimensional array of size (nVars,nBins) where nVars is defined as PLAG_SURF_STATS_LAST +nCont. It is of REAL type invoked as pPatch%surfPlag(iPatch)%vars(iVar,iBin).

5.2.6 PLAG communication datastructure t_border_data

(Rocflu Only)

The communication datastructure for PLAG with Rocflu is based on the pBorder infrastructure in ModBorder.F90. The communication buffers are of the form pBorder%plag%sendBuff and pBorder%plag%recvBuff. Separate buffers are used for the real and integer arrays.

```
MODULE ModBorder

  USE ModDataTypes

  IMPLICIT NONE

! ******************************************************************************
! Type definition
! ******************************************************************************

  TYPE t_border_data
    INTEGER :: sendRequest,sendRequestInt,tag,tagInt
    INTEGER, DIMENSION(:,:), POINTER :: recvBuffInt,sendBuffInt
    REAL(RFREAL), DIMENSION(:,:), POINTER :: recvBuff,sendBuff
  END TYPE t_border_data

  TYPE t_border
    INTEGER :: iBorder,iProc,iRegionGlobal,iRegionLocal
    INTEGER :: nCellsRecv,nCellsSend,nVertRecv,nVertSend,nVertShared
#ifdef PLAG
    INTEGER :: nPclsRecv,nPclsSend
#endif
    INTEGER, DIMENSION(:), POINTER :: icgRecv,icgSend,ivgRecv,ivgSend,ivgShared
#ifdef PLAG
    INTEGER, DIMENSION(:,:), POINTER :: iPclSend
#endif
    TYPE(t_border_data) :: mixt,spec,plag
  END TYPE t_border

END MODULE ModBorder
```

5.2.7 Datastructure Access for Rocflo

The following example illustrates how the particle datastructure is accessed for each region in Rocflo:

- t_plag

```
! Get dimensions ------------------------------------------------------------

  iLev  = region%currLevel
  iReg  = region%iRegionGlobal

  nPcls = region%levels(iLev)%plag%nPcls
  nCont = region%plagInput%nCont

! Set pointers ---------------------------------------------------------------

  pPlag   => region%levels(iLev)%plag

  pCv     => pPlag%cv
  pCvOld  => pPlag%cvOld
  pRhs    => pPlag%rhs
  pRhsSum => pPlag%rhsSum
  pCvPlagMass => pPlag%cvPlagMass
```

- t_tile_plag

```
! loop over all grid and patches levels --------------------------------------

  DO iLev=1,region%nGridLevels
    DO iPatch=1,region%nPatches

      pPatch => region%levels(iLev)%patches(iPatch)
      bcType =  pPatch%bcType
      nCont  =  region%plagInput%nCont

      IF ( bcType>=BC_INJECTION .AND. bcType<=BC_INJECTION+BC_RANGE ) THEN
         n1      = ABS(pPatch%l1end   -pPatch%l1beg   ) + 1
         n2      = ABS(pPatch%l2end   -pPatch%l2beg   ) + 1
         nTile   = n1*n2

         pTilePlag => pPatch%tilePlag

         pTilePlag%nCv = CV_TILE_LAST + nCont
         pTilePlag%nDv = DV_TILE_LAST

         nCv  = pTilePlag%nCv
         nDv  = pTilePlag%nDv

         ALLOCATE( pTilePlag%nPclsInjc(nTile),stat=errorFlag )

      ENDIF  ! bcType
    ENDDO    ! iPatch
  ENDDO   ! iLev
```

- t_buffer_plag

```
! Get dimensions ------------------------------------------------------------

  nCont        = regions(iReg)%plagInput%nCont
  nBuffSizeTot = regions(iReg)%plagInput%nPclsBuffTot
```

```
! Loop over all grid levels ------------------------------------------------

  DO iLev=1,regions(iReg)%nGridLevels

! - Set pointer and get dimensions -------------------------------------------

    pPlag  => regions(iReg)%levels(iLev)%plag
    nAiv = pPlag%nAiv
    nArv = pPlag%nArv
    nCv  = pPlag%nCv
    nDv  = pPlag%nDv
    nTv  = pPlag%nTv

    DO iPatch=1,regions(iReg)%nPatches

      pPatch  => regions(iReg)%levels(iLev)%patches(iPatch)
      bcType  =  pPatch%bcType
      lbound  =  pPatch%lbound
      iRegSrc =  pPatch%srcRegion

      n1      = ABS(pPatch%l1end   -pPatch%l1beg   ) + 1
      n2      = ABS(pPatch%l2end   -pPatch%l2beg   ) + 1
      nPatchSize  = n1*n2

      pBuffPlag => pPatch%bufferPlag

      IF ( (bcType>=BC_REGIONCONF .AND. bcType<=BC_REGIONCONF+BC_RANGE) .OR. &
           (bcType>=BC_TRA_PERI   .AND. bcType<=BC_TRA_PERI  +BC_RANGE) .OR. &
           (bcType>=BC_ROT_PERI   .AND. bcType<=BC_ROT_PERI  +BC_RANGE) ) THEN

        pBuffPlag%nBuffSizeTot = nBuffSizeTot

! - Allocate data for same-processor communication ---------------------------

        IF (regions(iRegSrc)%procid == global%myProcid) THEN
          ALLOCATE( pBuffPlag%aiv(nAiv,nBuffSizeTot),stat=errorFlag )

        END IF !regions

! - Allocate data for off-processor communication ----------------------------

        IF (regions(iRegSrc)%procid /= global%myProcid) THEN  ! other processor
          nBuffSizeI = nAiv*nBuffSizeTot
          nBuffSizeR = (nArv+4*nCv+nDv+nTv)*nBuffSizeTot
          nBuffSizeL = nBuffSizeTot

          pBuffPlag%nSendBuffTotI = nBuffSizeI
          pBuffPlag%nSendBuffTotR = nBuffSizeR

          pBuffPlag%nRecvBuffTotI = nBuffSizeI
          pBuffPlag%nRecvBuffTotR = nBuffSizeR

          ALLOCATE( pBuffPlag%sendBuffR(nBuffSizeR),stat=errorFlag )

      ELSE IF ((bcType>=BC_REGIONINT  .AND. bcType<=BC_REGIONINT +BC_RANGE) .OR. &
               (bcType>=BC_REGNONCONF .AND. bcType<=BC_REGNONCONF+BC_RANGE)) THEN
        CALL ErrorStop( global,ERR_UNKNOWN_BC,__LINE__ )  ! #### TEMPORARY ####

      ELSE
        NULLIFY(pBuffPlag%aiv)
        NULLIFY(pBuffPlag%arv)
        NULLIFY(pBuffPlag%cv)
        NULLIFY(pBuffPlag%dv)
        NULLIFY(pBuffPlag%tv)
```

```
      NULLIFY(pBuffPlag%rhs)
      NULLIFY(pBuffPlag%rhsSum)
      NULLIFY(pBuffPlag%aivOld)
      NULLIFY(pBuffPlag%arvOld)
      NULLIFY(pBuffPlag%cvOld)
      NULLIFY(pBuffPlag%sendBuffR)
      NULLIFY(pBuffPlag%sendBuffI)
      NULLIFY(pBuffPlag%sendBuffL)
      NULLIFY(pBuffPlag%recvBuffR)
      NULLIFY(pBuffPlag%recvBuffI)
      NULLIFY(pBuffPlag%recvBuffL)
    ENDIF     ! bcType

  ENDDO             ! iPatch

ENDDO   ! iLev
```

5.2.8 Datastructure Access for Rocflu

The following example illustrates how the particle datastructure is accessed for each region in Rocflu:

- t_plag

```
! Get dimensions ----------------------------------------------------------

  iReg  = pRegion%iRegionGlobal

  nPcls = pRegion%%plag%nPcls
  nCont = pRegion%plagInput%nCont

! Set pointers -------------------------------------------------------------

  pPlag   => pRegion%plag

  pCv     => pPlag%cv
  pCvOld  => pPlag%cvOld
  pRhs    => pPlag%rhs
  pRhsSum => pPlag%rhsSum
  pCvPlagMass => pPlag%cvPlagMass
```

- t_tile_plag

```
! ****************************************************************************
! Set pointers and variables
! ****************************************************************************

  pGrid => pRegion%grid

  nCont = pRegion%plagInput%nCont

! ****************************************************************************
! Allocate memory
! ****************************************************************************

  DO iPatch = 1,pGrid%nPatches
    pPatch => pRegion%patches(iPatch)
```

```
  IF ( (pPatch%bcType >= BC_INJECTION           ) .AND. &
       (pPatch%bcType <= BC_INJECTION + BC_RANGE) ) THEN
    pTilePlag => pPatch%tilePlag

    pPatch%tilePlag%nCv = CV_TILE_LAST + nCont
    pPatch%tilePlag%nDv = DV_TILE_LAST

    nCv = pTilePlag%nCv
    nDv = pTilePlag%nDv

    nTiles = pPatch%nBFaces

    ALLOCATE(pTilePlag%cv(nCv,nTiles),STAT=errorFlag)
    global%error = errorFlag
    IF (global%error /= ERR_NONE) THEN
      CALL ErrorStop(global, ERR_ALLOCATE,__LINE__,'pTilePlag%cv')
    END IF ! global%error

    ALLOCATE(pTilePlag%dv(nDv,nTiles),STAT=errorFlag)
    global%error = errorFlag
    IF (global%error /= ERR_NONE) THEN
      CALL ErrorStop(global, ERR_ALLOCATE,__LINE__,'pTilePlag%dv')
    END IF ! global%error

    ALLOCATE(pTilePlag%cvTileMass(nCont),STAT=errorFlag)
    global%error = errorFlag
    IF (global%error /= ERR_NONE) THEN
      CALL ErrorStop(global, ERR_ALLOCATE,__LINE__,'pTilePlag%cvTileMass')
    END IF ! global%error

    DO iCont = 1,nCont
      pTilePlag%cvTileMass(iCont) = CV_TILE_LAST + iCont
    END DO ! iCont
  END IF ! pPatch%bcType
END DO ! iPatch
```

Chapter 6

Implementation Details

6.1 Rocflo Specific Details

6.1.1 Search Algorithm for Cell Indices

For each particle in a computational cell, we follow an algorithm to search whether the particle has remained in this cell or moved to the adjoining ones. The steps of the search are outlined as follows:

- First search the computational cell from the previous RK step.
- Calculate at each computational cell surface the difference of the particle location and the cell surface node.
- Compute the dot product of this difference with the corresponding cell face normal.
- Set a flag either to zero or 1 if the product is negative or positive, respectively.
- Sum the values of this flag. If it is equal to 6, the particle remains in the computational cell.
- If this test fails, search the surrounding 27 cells using the same algorithm.
- Once the sum of this flag becomes equal to 6, the new particle indices have been obtained.
- Update the particle indices according to the search results.

The current implementation exits if the search algorithm fails beyond the surrounding 27 cells. This means that the timestep required is smaller that the one used. In this case, the user should decrease the CFL condition in `casename.inp` file.

6.1.2 Interpolation Algorithm for Fluid Properties at Particle Locations

The interpolation algorithm is determined by the value of `intrplMixtModel`.

For `intrplMixtModel = ZEROTH_ORDER`, a zeroth-order interpolation is invoked where the fluids properties at the current cell are prescribed at the particle location.

For `intrplMixtModel = FIRST_ORDER`, it is assumed that the grid is Cartesian, with the positive i-, j-, and k-axes aligned with the positive x-, y-, and z-axes respectively. A tri-linear interpolation is then invoked. Let (x_p, y_p, z_p) denote the particle coordinates. Let (x_c, y_c, z_c) be the cell center coordinates, and (ic, jc, kc) the indices, of the cell where the particle resides. The reference coordinate indices (ir, jr, kr) are determined as follows:

- If $x_p > x_c$ then $i_r = i_c$. Otherwise $i_r = i_c - 1$.
- If $y_p > y_c$ then $j_r = j_c$. Otherwise $j_r = j_c - 1$.
- If $z_p > z_c$ then $k_r = k_c$. Otherwise $k_r = k_c - 1$.

The fluid properties at the particle locations are then computed using trilinear interpolation functions as follows:

$$\begin{aligned} \xi_i &= \frac{x_p - x_{ir}}{x_{ir+1} - x_{ir}}; \\ \xi_j &= \frac{y_p - y_{jr}}{y_{jr+1} - y_{jr}}; \\ \xi_k &= \frac{z_p - z_{kr}}{z_{kr+1} - z_{kr}} \end{aligned} \tag{6.1}$$

$$\begin{aligned} u_f(x_p, y_p, z_p) = \; & (1-\xi_i)(1-\xi_j)(1-\xi_k) && u(ir, jr, kr) \\ + \; & \xi_i(1-\xi_j)(1-\xi_k) && u(ir+1, jr, kr) \\ + \; & (1-\xi_i)\xi_j(1-\xi_k) && u(ir, jr+1, kr) \\ + \; & \xi_i\xi_j(1-\xi_k) && u(ir+1, jr+1, kr) \\ + \; & (1-\xi_i)(1-\xi_j)\xi_k && u(ir, jr, kr+1) \\ + \; & \xi_i(1-\xi_j)\xi_k && u(ir+1, jr, kr+1) \\ + \; & (1-\xi_i)\xi_j\xi_k && u(ir, jr+1, kr+1) \\ + \; & \xi_i\xi_j\xi_k && u(ir+1, jr+1, kr+1) \end{aligned} \tag{6.2}$$

For `intrplMixtModel = SECOND_ORDER`, we implement what we call dual tet interpolation, which is valid for a general hexahedral mesh (provided it is not too badly skewed). The method is based on the interpretation of a hexahedron as the average of two tetrahedral partitions. Color the vertices of a hexahedron black and white by parity (i.e., so that each edge connects vertices of different colors), and consider the tetrahedra formed by each monochromatic set of vertices. The black tetrahedron, together with the four tetrahedra that each white vertex forms with the nearby face of the black tetrahedron, creates a partition of the hexahedron. The white tetrahedron creates another partition. We interpolate in the hexahedron by locating the particle in one tetrahedron from each partition, interpolating in each of these, then averaging the two results.

These are the steps that implement the above idea. The monochromatic tetrahedra are referred to as big tetrahedra. We initialize (ir, jr, kr) to be the indices of the cell where the particle resides.

- Step 1. Octant search:
  The six mesh edges joining the cell center (ir, jr, kr) to adjacent cell centers form eight octants, which are separated from each other by 12 planar surfaces. By determining on which side of each of these 12 surfaces the particle lies, we may deduce which octant the particle is in. We perform the tests in an order that tends to locate the particle quickly (an average of 4.25 tests for a Cartesian grid, increasing with the grid's skewness).

- Step 2. Near tet search:
  Let (id, jd, kd) denote the octant found in step 1, where each coordinate is ± 1. We now test whether the particle is on the same side of the plane determined by $(ir + id, jr, kr)$, $(ir, jr + jd, kr)$, and $(ir, jr, kr + kd)$ as (ir, jr, kr) is. If so, then we've located the particle within a tetrahedron, and we go to step 4.

- Step 3. Big tet search:
  We test whether the particle is on the same side of the plane determined by $(ir + id, jr + jd, kr + kd)$, $(ir, jr + jd, kr)$, and $(ir, jr, kr + kd)$ as $(ir + id, jr, kr)$ is. If not, then we reset (ir, jr, kr) to $(ir, jr + jd, kr + kd)$ and go to step 1. We perform two more such tests, on the remaining two faces of the big tetrahedron. If either fails, we go to step 1 (with (ir, jr, kr) reset appropriately). Otherwise, the particle is in the big tetrahedron.

- Step 4. Tetrahedral interpolation:
  Now that the particle is located within a tetrahedron, with vertices $\mathbf{V}_1$,$\mathbf{V}_2$,$\mathbf{V}_3$, and $\mathbf{V}_4$, let $\mathbf{v}_j = \mathbf{V}_j - \mathbf{V}_4$ for $j = 1$ to 3. Let the particle position be $\mathbf{X}$, and $\mathbf{x} = \mathbf{X} - \mathbf{V}_4$. We then solve the equation
  $$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + c_3\mathbf{v}_3 = \mathbf{x} \tag{6.3}$$
  for c_1, c_2, and c_3, and let $c_4 = 1 - (c_1 + c_2 + c_3)$. Quantities at the particle location are then determined as a linear combination of the values at $\mathbf{V}_1$ through $\mathbf{V}_4$, with coefficients c_1 through c_4.

- Step 5. Opposite parity:
  Repeat steps 1 through 4 with an initial cell center of opposite parity (we use $(ir+id, jr, kr)$). Average the results of the two interpolations.

6.1.3 Wall Reflection Algorithm

During the particle evolution, situations arise where a particle may "hit" a computational boundary. To be able to account for such situations, an elastic bounce algorithm has been devised. We search in a layer near the computational boundaries with the following boundary condition types, `BC_SLIPWALL,BC_NOSLIPWALL,BC_SYMMETRY,BC_INJECTION`. The boundary conditions are an Euler wall, a Navier-Stokes wall, a symmetry plane, and an injection wall, respectively. For the particle in a cell layer around the boundaries, we apply a search algorithm similar to the search algorithm for the particle indices. The steps are outlined as follows:

- Calculate for each particle whose indices lie on the boundaries of the computational block the difference of the particle location and the cell surface node.
  $$pmf(1:3) = ppos(1:3) - cgfcx(1:3) \tag{6.4}$$

where *ppos* are the particle coordinates and $cgfcx$ are the coordinates of the centers of the cell faces.

- Compute the dot product of this difference with the corresponding cell face normal.

$$dpx = dot\_product(cgnx, pmf) \tag{6.5}$$

where *cgnx* are the coordinates of the normal vector to the cell surfaces.

- Let *sgn* be 1 if the boundary is at i, j, or $k = 1$, and -1 otherwise (i.e., at $i = ni$, $j = nj$, or $k = nk$). If the product $dpx * sgn$ is negative, the particle has crossed the boundary, so impose reflective conditions: subtract from the velocity twice its normal component; similarly for position.

$$\begin{aligned} dpv &= dot\_product(cgnx, pvel) \\ dfac &= -2/dot\_product(cgnx, cgnx) \\ ppos(1:3) &= ppos(1:3) + dpx * dfac * cgnx(1:3) \\ pvel(1:3) &= pvel(1:3) + dpv * dfac * cgnx(1:3) \end{aligned} \tag{6.6}$$

- Loop over the six (6) boundary sides.

6.1.4 Assumptions

The following assumptions are imposed on Rocpart module:

- The same RK coefficients, α, are used for evolving the gas flow solver and the Lagrangian particle solver.
- Zero-order interpolation for the mixture properties is currently available. This assumption stems from 2-way coupling restrictions where the same approach has to be invoked when the particles back-influence the mixture field.
- Standard drag laws are invoked for the particle velocity update equation.
- Search algorithm for cell indices where particles reside is restricted to the 27 neighboring cells, i.e ± 1 around each cell.
- Current parallel implementation permits decomposition in any coordinate direction.

6.2 Rocflu Specific Details

6.2.1 Search Algorithm for Cell Indices

A trajectory-based algorithm has been designed to permit the seamless tracking of particles on mixed unstructured mesh. The mechanics of this algorithm is presently described in a paper to be published (see Haselbacher, A., Najjar, F. & Ferry, J., 2006). The routine where the search algorithm is implemented is found in PLAG_RFLU_FindCellsTraj. Several options are also available

mainly a brute force, an octree search and a known-vicinity search. However, these options are only useful for testing and performance analysis. They do not handle wall reflection.

The fundamental operation of the present particle-localization algorithm is the computation of the intersection of the particle trajectory with the faces of the cell which contains the particle. The following subsections describe the computation of the intersection of the trajectory with planar and non-planar faces.

6.2.1.1 Planar Faces

The problem of finding the intersection of the particle trajectory with a planar face can be abstracted as determining the intersection of a ray $\mathbf{t}$ anchored at the point $\mathbf{r}_P$,

$$\mathbf{r}(\alpha) = \mathbf{r}_P + \alpha\mathbf{t}, \tag{6.7}$$

with a plane specified by the normal vector $\mathbf{n}$ and anchored at the point $\mathbf{r}_C$,

$$(\mathbf{r} - \mathbf{r}_C) \cdot \mathbf{n} = 0. \tag{6.8}$$

Substituting Eq. (6.8) into Eq. (6.7) gives the distance between the intersection point $\mathbf{r}_I$ and $\mathbf{r}_P$ as

$$\alpha_I = \frac{(\mathbf{r}_C - \mathbf{r}_P) \cdot \mathbf{n}}{\mathbf{t} \cdot \mathbf{n}}. \tag{6.9}$$

Note that it is neither necessary to solve a linear system for the intersection point nor to compute the intersection distance from the square root of the summed squares of coordinate differences.

It is instructive to consider the meaning of the numerator and denominator of the right-hand side of Eq. (6.9) in the context of particle tracking. The numerator is the signed normal distance between the plane and the particle position; it is positive if the particle is located in the cell. The denominator indicates the orientation of the trajectory relative to the face normal; if the denominator is positive (negative), the particle is moving toward (away from) the face, and if it is zero, no intersection is possible.

6.2.1.2 Non-Planar Faces

Non-planar faces can be treated if Eq. (6.8) is replaced by the appropriate equation. For example, the parametric equation describing a bilinear patch can be written as

$$\mathbf{r}(u, v) = uv\mathbf{a} + u\mathbf{b} + v\mathbf{c} + \mathbf{d}, \tag{6.10}$$

where $0 \leq u \leq 1$ and $0 \leq v \leq 1$ are parameters, $\mathbf{a} = \mathbf{r}_{00} - \mathbf{r}_{10} + \mathbf{r}_{11} - \mathbf{r}_{01}$, $\mathbf{b} = \mathbf{r}_{10} - \mathbf{r}_{00}$, $\mathbf{c} = \mathbf{r}_{01} - \mathbf{r}_{00}$, $\mathbf{d} = \mathbf{r}_{00}$, and $\mathbf{r}_{ij}$ are the position vectors of four non-coplanar points, i.e., $\mathbf{a} \neq \mathbf{0}$. The intersection point(s) of a ray and a bilinear patch can then again be computed from Eqs. (6.10) and (6.7). Note that a ray may intersect a bilinear patch in one location, two locations, or not at all. Ramsey et al. (2004) recently presented an efficient and robust algorithm to compute the intersection point(s) of a ray and a bilinear patch. Here we present a modification of this algorithm which is more efficient.

The intersection point with the ray given by Eq. (6.7) is given by

$$uv\mathbf{a} + u\mathbf{b} + v\mathbf{c} + \tilde{\mathbf{d}} = \alpha_I\mathbf{t}, \tag{6.11}$$

where $\tilde{\mathbf{d}} = \mathbf{d} - \mathbf{r}_P$. The intersection distance is obtained immediately from

$$\alpha_I = \left(uv\mathbf{a} + u\mathbf{b} + v\mathbf{c} + \tilde{\mathbf{d}} \right) \cdot \mathbf{t} \tag{6.12}$$

Substituting Eq. (6.12) into Eq. (6.11) gives

$$uv\mathbf{a}^{\perp} + u\mathbf{b}^{\perp} + v\mathbf{c}^{\perp} + \tilde{\mathbf{d}}^{\perp} = \mathbf{0}, \tag{6.13}$$

where the superscript $\perp$ denotes the component normal to the ray $\mathbf{t}$, e.g., $\mathbf{a}^{\perp} = \mathbf{a} - (\mathbf{a} \cdot \mathbf{t})\,\mathbf{t}$. Equation (6.13) can be expressed as

$$uva_{xz}^{\perp} + ub_{xz}^{\perp} + vc_{xz}^{\perp} + \tilde{d}_{xz}^{\perp} = 0, \tag{6.14}$$

$$uva_{yz}^{\perp} + ub_{yz}^{\perp} + vc_{yz}^{\perp} + \tilde{d}_{yz}^{\perp} = 0, \tag{6.15}$$

where the subscripts denote differences of components, i.e., $a_{xz}^{\perp} = a_z^{\perp} - a_x^{\perp}$. Solving Eq. (6.14) for u and substituting into Eq. (6.15) leads to

$$\left(a_{xz}^{\perp} c_{yz}^{\perp} - a_{yz}^{\perp} c_{xz}^{\perp} \right) v^2 + \left(a_{xz}^{\perp} \tilde{d}_{yz}^{\perp} - a_{yz}^{\perp} \tilde{d}_{xz}^{\perp} + b_{xz}^{\perp} c_{yz}^{\perp} - b_{yz}^{\perp} c_{xz}^{\perp} \right) v + b_{xz}^{\perp} \tilde{d}_{yz}^{\perp} - b_{yz}^{\perp} \tilde{d}_{xz}^{\perp} = 0. \tag{6.16}$$

For any root $0 \leq v \leq 1$, the corresponding u can be computed from Eqs. (6.14) or (6.15) and if $0 \leq u \leq 1$, the associated intersection distance is determined from Eq. (6.12). Otherwise, the ray does not intersect the bilinear patch.

If the ray intersects the bilinear patch in more than one location, then the intersection point with the smaller intersection distance should be taken in principle, as discussed above. However, if the distance which remains to be traveled exceeds the larger intersection distance, then the intersections with that face can be ignored to improve efficiency. This simplification is possible because the ray, after exiting the cell through the intersection point with the smaller intersection distance, would reenter the same cell through the intersection point with the larger intersection distance.

6.2.2 Interpolation Algorithm for Fluid Properties at Particle Locations

The interpolation algorithm is determined by the value of `intrplMixtModel`. The current support is for piecewise constant, `intrplMixtModel = ZEROTH_ORDER` and piecewise linear, `intrplMixtModel = ZEROTH_FIRST`. A zeroth-order interpolation is invoked when a first-order spatial discretization scheme is invoked; while the first-order interpolation can be only used with a second-order spatial discretization scheme. The general formulation to compute a mixture variable at a particle location based on the values at the cell centroid for these two interpolations is:

$$\phi(\mathbf{x}_p) = \phi(\mathbf{x}_{cc}) + \overrightarrow{\mathbf{r}} \cdot \overrightarrow{\nabla \phi} \tag{6.17}$$

where $\overrightarrow{\mathbf{r}}$ is the vector representing the distance between the particle location and the cell centroid and $\nabla\phi$ is the gradient vector of the variable.

To compute the mixture properties at the locations of a given droplet, either piecewise constant or piecewise linear interpolation is used. These interpolations result in first- or second-order accurate estimates of the mixture properties, respectively. Assuming that cell i contains the droplet location $\mathbf{r}^p$, the piecewise constant interpolation can be expressed as

$$\phi(\mathbf{r}^p) = \phi_i \tag{6.18}$$

where ϕ represents a mixture variable to be interpolated such as the density, velocity, pressure and temperature. Piecewise linear interpolation is given by

$$\phi(\mathbf{r}^p) = \phi_i + (\boldsymbol{\delta}\phi)_i \cdot \Delta\mathbf{r}_i^p \tag{6.19}$$

where $(\boldsymbol{\delta}\phi)_i$ is the discrete gradient of ϕ at the centroid of cell i and $\Delta\mathbf{r}_i^p = \mathbf{r}^p - \mathbf{r}_i$ is the relative position vector from the centroid of cell i to the droplet location.

6.2.3 Wall Reflection Algorithm

During the particle evolution, situations arise where a particle may "hit" a computational boundary. To be able to account for such situations, an elastic bounce algorithm has been devised. The tracking algorithm traps when a particle has passed through any of the following boundary condition types, `BC_SLIPWALL,BC_NOSLIPWALL,BC_SYMMETRY,BC_INJECTION,BC_VIRTUAL`. The boundary conditions are an Euler wall, a Navier-Stokes wall, a symmetry plane, and an injection wall, respectively. The steps are similar to those outlined for the corresponding Rocflu design. The routine where the wall reflection is implemented is found in PLAG_ReflectParticleData

6.2.4 Assumptions

The following assumptions are imposed on Rocpart module:

- The same RK coefficients, α, are used for evolving the gas flow solver and the Lagrangian particle solver. Only the low-storage three-stage Runge-Kutta algorithm is active in Rocflu with Rocpart being active.
- Zero and first-order interpolations for the mixture properties are currently available.
- Standard drag laws are invoked for the particle velocity update equation.
- Search algorithm for cell indices where particles reside is general to permit any particle motion in the computational domain.

Chapter 7

Parallel Implementation

7.1 Rocflo Specific Details

7.1.1 Basic Algorithm

Four distinct levels for communication of the Lagrangian particles across region boundaries exist and include:

- (i) particles moving across regions on the same processor.
- (ii) particles moving across adjacent regions on different processors.
- (iii) particles crossing regions connected through edges or corners on the same processor.
- (iv) particles crossing regions connected through edges or corners on different processors.

Steps (i) and (iv) require memory copy across regions, while Steps (ii) and (iii) are completed via MPI calling sequences. The current capability of Rocpart has Steps (i)-(iv) fully implemented. The main difference for Rocpart with Eulerian solvers such as Rocflo, and Rocsmoke, is that the buffer size changes dynamically at each timestep. While the gas solvers are aware initially of the array sizes to communicate, Rocpart has to determine that buffer size as the computation evolves.

7.1.2 Adjacent Regions

The main kernel for communications pertaining to Steps (i) and (ii) is as follows:

- obtain patch buffer size (`PLAG_PatchGetBufferSize`)
- load data buffer and reshuffle particle datastructure (`PLAG_PatchLoadDataBuffers`)
- exchange datastructure for regions on same processor (`PLAG_BoundaryConditionsSet`)
- communicate buffer data for adjacent regions on different processors (`PLAG_PatchBufferSendRecv`)
  - ▶ send buffer size (`PLAG_BufferSizeSend`)
  - ▶ receive buffer size (`PLAG_BufferSizeRecv`)

- wait for data being received by other processors (PLAG_ClearSizeSendRequest)
- send buffer data (PLAG_BufferDataSend)
- receive buffer data (PLAG_BufferDataRecv)
- wait for data being received by other processors (PLAG_ClearDataSendRequests)

7.1.3 Regions Communicating across Their Edges and Corners

The main kernel for communications pertaining to Step (iii) is as follows (PLAG_CECellsWrapper):

- determine in the initial step the geometry-based metrics (cells PLAG_RFLO_setMetrics) and update these values in the corner and edge cells (PLAG_CECellsFaceCentroids) and (PLAG_CECellsFaceVectors)
- obtain buffer size (PLAG_CECellsGetBufferSize)
- load data buffer and reshuffle particle datastructure (PLAG_CECellsLoadDataBuffers)
- exchange datastructure for regions on same processor (PLAG_CECellsExchange)

7.2 Rocflu Specific Details

Two distinct levels for communication of the Lagrangian particles across region borders exist and include:

- (i) particles moving across regions on the same processor.
- (ii) particles moving across adjacent regions on different processors.

Step (i) require a memory copy across regions, while Step (ii) is completed via MPI calling sequences. The current capability of Rocpart has Steps (i)-(ii) fully implemented. In Rocflu each domain has a set of borders used for communications.

The main difference for Rocpart with the Eulerian solvers such as Rocflu, and Rocspecies, is that the buffer size changes dynamically at each timestep. While the gas solvers are aware initially of the array sizes to communicate, Rocpart has to determine that buffer size as the computation evolves.

As the particles are being evolved and tracked through the finite-volume cells it is moving, the algorithm keeps track of the number of particle passing through the communicating borders. A particle-specific flag (aiv(AIV_PLAG_STATUS)) is set to PLAG_STATUS_COMM). Refer to PLAG_RFLU_ModFindCells.F90) module.

7.2.1 Basic Algorithm

The main module for the parallel algorithm is PLAG_RFLU_ModComm.F90, and the corresponding kernel for communications is as follows:

- find if any region need to communicate particles and exit if null (PLAG_RFLU_TotalnPclsComm)

- initialize receive counters (PLAG_RFLU_InitRecvCounters)
- send counters (PLAG_RFLU_ISendCounters)
- copy counters between regions whose borders are on same processor (PLAG_RFLU_CopyCounters)
- create send data buffers (PLAG_RFLU_CreateBuffersSend)
- fill send data buffers (PLAG_RFLU_LoadBuffersSend)
- receive counters (PLAG_RFLU_RecvCounters)
- clear requests for counters (PLAG_RFLU_ClearRequestWrapper)
- create receive data buffers (PLAG_RFLU_CreateBuffersRecv)
- send communication buffers (PLAG_RFLU_ISendData)
- copy data buffers for regions whose borders are on same processor (PLAG_RFLU_CopyData)
- update datastructure on sending side (PLAG_UpdateDataStruct)
- receive data buffers (PLAG_RFLU_RecvData)
- clear requests for data buffers (PLAG_RFLU_ClearRequestWrapper)
- unload receive buffers (PLAG_RFLU_UnloadBuffersRecv)
- deallocate memory for send buffers (PLAG_RFLU_DestroyBuffersSend), and receive buffers (PLAG_RFLU_DestroyBuffersRecv)
- reallocate memory (PLAG_ReallocMemWrapper)
- initialize send counters (PLAG_RFLU_InitSendCounters)
- continue tracking particles if remaining trajectory distance is not zero
- update datastructure on receiving side (PLAG_UpdateDataStruct)

Chapter 8

Rocstar Integration

8.1 Rocflo Specific Details

The Rocpart integration into Rocstar is isolated to two routines (`PLAG_InitGenxInterface` and `PLAG_SetSizeGenx`) where the datastructure gets registered with Roccom. Current efforts are to generalize the boundary conditions to account for the Lagrangian particles. Eventhough the size of the Lagrangian particles evolves dynamically during the course of the simulations, the restart files within Rocstar 3.0 are based on a variable fixed size of `nPcls`.

The main kernel in `PLAG_initGenxInterface` is as follows:

- (i) register the surface tile data as part of Rocflo surface data. This includes the normal momentum, the energy, the mass and time factor variable.
- (ii) register the volume data for Rocpart. This includes `aiv`, `cv`, `arv`, and `dv` variables as well as the total number of particles in each region and the next particle id number.

8.2 Rocflu Specific Details

Rocpart version running with Rocflu is currently under development to be Rocstar-aware.

Chapter 9

Installation and Compilation

9.1 Installation

9.1.1 Installation of Rocfluid

The procedure outlined below assumes that Rocfluid is to be installed either from the CSAR CVS repository or from a `gzip`ped `tar` file.

9.1.1.1 Installation from CVS Repository

To be able to access the CSAR's CVS repository, set the `CVSROOT` environment variable to (taking the `bash` shell as an example)

```
export CVSROOT=:pserver:<username>@galileo.csar.uiuc.edu:/cvsroot
```

and either open a new terminal or type

```
[user@machine ~]$ source .bashrc
```

Then type

```
[user@machine ~]$ cvs login
```

and hit the `Enter` key at the prompt.

Now move into the directory where you want to install Rocflo. In the following, this is assumed to be `directory`. Then type

```
[user@machine ~/directory]$ cvs co Rocstar/RocfluidMP
```

which will check out the source code for Rocflo from the repository.

Assuming the checkout command has completed successfully, you are now ready to compile the code for serial computations, and you can proceed to Sec. **??**.

9.1.1.2 Installation from .tar.gz File

Move into the directory where you want to install Rocflo. In the following, this is assumed to be `directory`. Move or copy the `gzip`ped `tar` file, assumed to be `<file>.tar.gz` in the following, into `directory`. Then type

```
[user@machine ~/directory]$ gzip -d <file>.tar.gz
[user@machine ~/directory]$ tar -xvf <file>.tar
```

which will unpack the source code.

Assuming these commands to have completed successfully, you are now ready to compile the code for serial computations, and you can proceed to Sec. 9.2 or Sec. 9.3 to compile with Rocflo or Rocflu, respectively.

9.2 Compilation with Rocflo

9.2.1 Overview of Compilation Process

The compilation process for Rocfluid is automatic in the sense that the `Makefile`s determine the machine type and set the suitable compilation options. If you intend to run on IBM, Linux, SGI, or Sun machines, you do not need to modify any `Makefile`s. If you intend to run on other machines, you will need to create your own `Makefile`.

The compilation process consists of two parts. The first part is the actual computation, as described below. The output of the compilation process are several executables:

rfloprep The preprocessing module of Rocflo.

rflomp The flow solver.

rflopost The postprocessing module of Rocflo. (Only compiled if compile with `POST=1`, see below.)

plagpost The postprocessing module of Rocpart.(Only compiled if compile with `POST=1`, see below.)

The second part consists of copying these executables into your `$(HOME)/bin` directory by typing:

```
[user@machine ~/directory]$ gmake RFLO=1 PLAG=1 install
```

9.2.2 Description of Compilation Options

To compile Rocflo, type the following at the prompt:

```
[user@machine ~/directory]$ gmake RFLO=1 PLAG=1 <options>
```

where the currently supported `<options>` are any of the following:

PLAG_DEBUG=(0|1) Deactivates or activates debugging compiler options. Specifying DEBUG=0, or leaving out the option altogether, means that no debugging options will be used. Specifying PLAG_DEBUG=1 will activate debugging options.

PLAG_MPIDEBUG=(0|1) Deactivates or activates debugging compiler options under MPI construct. Specifying DEBUG=0, or leaving out the option altogether, means that no debugging options will be used. Specifying PLAG_MPIDEBUG=1 will activate debugging options.

POST=(0|1) Deactivates or activates compilation of the postprocessing module rplagpost. Specifying POST=0, or leaving out the option altogether, means that rplagpost will not be compiled. Specifying POST=1 will lead to compilation of rplagpost.

To compile Rocflowith particles using MPI, type the folowing at the prompt:

```
[user@machine ~/directory]$ gmake RFLO=1 PLAG=1 MPI=1
```

To compile Rocflowith particles and smoke, type the folowing at the prompt:

```
[user@machine ~/directory]$ gmake RFLO=1 PLAG=1 PEUL=1
```

To compile Rocflowith particles, smoke and turbulence, type the folowing at the prompt:

```
[user@machine ~/directory]$ gmake RFLO=1 PLAG=1 PEUL=1 TURB=1
```

9.3 Compilation with Rocflu

9.3.1 Overview of Compilation Process

The compilation process for Rocfluid is automatic in the sense that the Makefiles determine the machine type and set the suitable compilation options. If you intend to run on IBM, Linux, SGI, Macintosh, or Sun machines, you do not need to modify any Makefiles. If you intend to run on other machines, you will need to create your own Makefile.

The compilation process consists of two parts. The first part is the actual computation, as described below. The output of the compilation process are several executables:

rflumap, rflupart, rfluinit The preprocessing modules of Rocflu.

rflump The flow solver.

rflupost The postprocessing module of Rocflu.

The second part consists of copying these executables into your $(HOME)/bin directory by typing:

```
[user@machine ~/directory]$ gmake RFLO=1 PLAG=1 install
```

9.3.2 Description of Compilation Options

To compile Rocflo, type the following at the prompt:

```
[user@machine ~/directory]$ gmake RFLU=1 PLAG=1
```

Chapter 10

Execution

This chapter contains detailed information on the command-line arguments and input and output files of Rocflo, plagpost, and Rocflu.

10.1 Rocflo

10.1.1 Command-Line Arguments

For serial computations, Rocflo is invoked by typing

```
rflomp <casename> <verbosity>
```

where

`<casename>` is a character string used to label the input and output files.

`<verbosity>` is an integer indicating the desired verbosity level of Rocflo. The verbosity level can take the following values:

- `0` No output. Rocflo will not write any information to standard output.
- `1` Low level of output. Rocflo will write some information to standard output.
- `2` High level of output. Rocflo will write detailed information to standard output.

For parallel computations, Rocflo is invoked by typing

```
mpirun -np <number_of_processors> rflomp <casename> <verbosity>
```

10.1.2 Input Files

The following input files are read by Rocflo:

- An input file called `<casename>.inp`.
- A grid file in Rocflo format.

- A boundary condition file. The name of the file is <casename>.bc.
- A topology file. The name of the file is <casename>.top.
- A flow solution file in Rocflo format. The name of the flow solution file is specified for an unsteady computation (refer to Rocflo User's Guide).
- A particle solution file in Rocpart format. The name of the particle solution file is <casename>.plag_sola_0.00000E+00 for ASCII formatted file
  and <casename>.plag_solb_0.00000E+00 for binary formatted file.

10.1.3 Output Files

The following input files are written by Rocflo:

- A flow solution file in Rocflo format.
- A particle solution file in Rocpart format. The name of the particle solution file is <casename>.plag_sola_<stamp> for ASCII formatted file
  and <casename>.plag_solb_<stamp> for binary formatted file.

10.2 plagpost with Rocflo

10.2.1 Command-Line Arguments

For serial computations, plagpost is invoked by typing

```
plagpost <casename> <time> <format>
```

where

<casename> is a character string used to label the input and output files.

<time> is a variable indicating the time from which the solution file is to be read.

<format> is an integer indicating the type of output. <format> can take the following values:

1 Write solution output file in generic binary format.

2 Write solution output file in Tecplot binary format.

3 Write solution output file in Tecplot ASCII format.

Currently only option 3 is supported

10.2.2 Input Files

No input files are currently written by rplagpost:

10.2.3 Output Files

The following output files are written by rplagpost:

- A file in Tecplot format called <casename>.plag_<stamp>.plt.

10.3 Rocflu

10.3.1 Command-Line Arguments

For serial computations, Rocflu is invoked by typing

```
rflumap -c <casename> -m <type> -p <procs> -r <regions> -v <verbosity>

rflupart -c <casename> -v <verbosity>

rfluinit -c <casename> -v <verbosity>

rflump -c <casename> -v <verbosity>

rflupost -c <casename> -s <timestamp> -v <verbosity>
```

For parallel computations, Rocflu is invoked by typing

```
rflumap -c <casename> -m <type> -p <procs> -r <regions> -v <verbosity>

rflupart -c <casename> -v <verbosity>

rfluinit -c <casename> -v <verbosity>

mpirun -np <procs> rflump -c <casename> -v <verbosity>

rflupost -c <casename> -s <timestamp> -v <verbosity>
```

where

<casename> is a character string used to label the input and output files.

<verbosity> is an integer indicating the desired verbosity level of Rocflu. The verbosity level can take the following values:

0 No output. Rocflu will not write any information to standard output.

1 Low level of output. Rocflu will write some information to standard output.

2 High level of output. Rocflu will write detailed information to standard output.

<timestamp> is a real indicating the desired timestamp to create the visualization files.

<nprocs> is the number of processors.

<regions> is the number of regions.

The preparation tools (rflumap,rflupart, & rfluinit) generate all the pertinent files to perform a calculation with particles that are ran by rflump. The postprocessing tool (rflupost) creates the pertinent files for visualization purposes. Further details can be found in the Rocflu Developer's and User's Guides.

10.3.2 Input Files

The following input files are read by Rocflu:

- An input file called <casename>.inp.
- A grid file in Rocflu format.
- A boundary condition file. The name of the file is <casename>.bc.
- A map file. The name of the file is <casename>.map.
- A suite of communication files. The name of the file is <casename>.com and <casename>.rnm.
- A flow solution file in Rocflu format. The name of the flow solution file is specified for an unsteady computation (refer to Rocflu User's Guide).
- A particle dimension file in Rocpart format. The name of the particle solution file is <casename>.pdim_00000_0.00000E+00. It is always in ASCII format.
- A particle solution file in Rocpart format. The name of the particle solution file is <casename>.plag_sola_00000_0.00000E+00 for ASCII formatted file and <casename>.plag_sol_00000_0.00000E+00 for binary formatted file.

10.3.3 Output Files

The following input files are written by Rocflu:

- A flow solution file in Rocflu format.
- A particle dimension file in Rocpart format. The name of the particle solution file is <casename>.pdim_region>_<stamp>. It is always in ASCII format.
- A particle solution file in Rocpart format. The name of the particle solution file is <casename>.plag_sola_<region>_<stamp> for ASCII formatted file and <casename>.plag_sol_<region>_<stamp> for binary formatted file.