OhHelp Library Package
for Scalable Domain-Decomposed PIC Simulation*

Hiroshi Nakashima
(ACCMS, Kyoto University)

2009/08/21

Abstract

This document describes the usage of a C-code library package named OhHelp for domain-decomposed Particle-in-Cell (PIC) simulations. The library has the following three layers. Level-1 code provides a load-balancer function which examines whether particles are distributed among computation nodes (MPI processes) in a well-balanced manner, reforms the configuration of particle assignment to each node if necessary, and tells you how to move particles among nodes. In Level-2 code, the load balancer function is also capable to move particles among nodes by MPI functions for you. In addition, Level-3 code has various useful functions for domain-decomposed simulations such as for exchanging boundary values of electromagnetic fields associated to decomposed subdomain.

*This file has version number v0.9, last revised 2009/08/21.
# Contents

1 Introduction ................................................. 4

2 OhHelp Algorithm ........................................... 5
   2.1 Overview and Definitions ................................. 5
   2.2 Secondary Subdomain Assignment ......................... 6
   2.3 Checking and Keeping Local Balancing ................... 7

3 OhHelp Library ............................................. 11
   3.1 Library Layers .......................................... 11
   3.2 Applying OhHelp to PIC Simulators ....................... 11
      3.2.1 Duplication of Data Structures ...................... 13
      3.2.2 Duplication of Computation .......................... 14
      3.2.3 Addition of Collective Communications ............... 16
      3.2.4 Attachment of Load Balancer ........................ 17
   3.3 Dimension of Simulated Space ............................ 18
   3.4 Level-1 Library Functions ............................... 18
      3.4.1 oh1_init() ........................................ 19
      3.4.2 oh1_transbound() .................................. 25
      3.4.3 oh1_broadcast() ................................... 27
      3.4.4 oh1_all_reduce() .................................. 28
      3.4.5 oh1_reduce() ....................................... 29
   3.5 Level-2 Library Functions ............................... 30
      3.5.1 Particle Data Type .................................. 30
      3.5.2 oh2_init() ........................................ 31
      3.5.3 oh2_max_local_particles() ........................... 32
      3.5.4 oh2_transbound() ................................... 33
      3.5.5 oh2_inject_particle() ............................... 34
   3.6 Level-3 Library Functions ............................... 34
      3.6.1 oh3_init() .......................................... 35
      3.6.2 oh13_init() ......................................... 49
      3.6.3 oh3_transbound() ................................... 50
      3.6.4 oh3_map_particle_to_neighbor() ..................... 51
      3.6.5 oh3_map_particle_to_subdomain() .................... 52
      3.6.6 oh3_bcast_field() .................................. 54
      3.6.7 oh3_allreduce_field() ............................... 55
      3.6.8 oh3_reduce_field() .................................. 56
      3.6.9 oh3_exchange_borders() .............................. 56
   3.7 Particle Injection and Removal ......................... 58
   3.8 Statistics ................................................ 59
      3.8.1 Timing Statistics Keys and Header File oh_stats.h ... 59
      3.8.2 Arguments of oh1_init() for Statistics .............. 61
      3.8.3 oh1_init_stats() .................................... 62
      3.8.4 oh1_stats_time() .................................... 62
      3.8.5 oh1_show_stats() .................................... 63
      3.8.6 oh1_print_stats() .................................... 63
   3.9 Verbose Messaging ........................................ 65
   3.10 Aliases of Functions .................................... 66
   3.11 Sample Code ............................................. 67
1 Introduction

Particle-in-Cell (PIC) simulations have played an indispensable role in theoretical and practical research of high-energy physics, space plasma physics, cloud modeling, combustion engineering, and so on, since early 1980’s. In typical PIC simulations, a huge number of charged particles interact with electromagnetic fields mapped onto a large number of grid points, governed by Maxwell’s equations and the Lorentz force law. These hugeness and largeness of the simulation essentially require to parallelize the computation not only for efficient execution but also for feasible implementation on distributed memory systems which are the majority of modern supercomputers. That is, the simulation has to be decomposed almost equally so that good load balancing is achieved and, more importantly, each decomposed subproblem is accommodated by a local memory of limited capacity. This almost-equal decomposition is a necessary condition to make the simulation scalable so that we fully utilize larger scale systems with nearly stable efficiency by enlarging the problem size proportionally to the system size.

However, this necessary condition is satisfied neither by simple particle-decomposed simulations, by also simple static domain-decomposed ones, nor even by sophisticated dynamic domain-decomposed simulations, because a process in these conventional methods would have too large (sub)domain or too many particles. Therefore, we have proposed a new domain-decomposed PIC simulation method named OhHelp[1] which is scalable in terms of the number of particles as well as the domain size. Its problem decomposition and load balancing mechanisms are outlined as follows.

1. The space domain is equally partitioned to assign each subdomain to each node as its primary subdomain.

2. If one or more subdomains have too many particles, i.e., more than average plus a certain tolerance, every but one node is responsible for another subdomain which has particles more than average as its secondary subdomain.

3. A part of particles in the secondary subdomain of a node are assigned to the node so that no nodes have too many particles.

Since a node has to have at most two subdomains, OhHelp is scalable with respect to the domain size. As for the number of particles, OhHelp keeps its excess over the per-node average less than the tolerance by dynamically rearranging the secondary subdomain assignment and thus also achieves good scalability.

In the rest of this document, we describe OhHelp and its library as follows. In the next Section 2, OhHelp algorithm is explained more detailedly. Then Section 3, the heart of this document, describes API of the OhHelp library so that you incorporate OhHelp into your own PIC simulator.

References

2 OhHelp Algorithm

2.1 Overview and Definitions

As shown in Figure 1, OhHelp simply partitions the simulated $D$-dimensional space domain ($D \leq 3$) into (almost) equal-size $N$ subdomains and assigns each subdomain $n$ ($n \in [0, N-1]$) to each of $N$ (MPI) processes, or computation node, whose MPI rank, or identifier, is also $n$, as its primary subdomain. In the figure, non-italic black numbers are the identifiers of nodes and also those of primary subdomains assigned to them. Each node $n$ is responsible for its primary subdomain $n$, and also all the particles in it if the numbers of those primary particles in subdomains are balanced well, or more specifically, if the number of particles $P_n$ in a subdomain $n$ satisfies the following inequality for all $n$,

$$P_n \leq (P/N)(100 + \alpha)/100 \equiv P_{\text{max}}$$

(1)

where $P$ is the total number of particles and $\alpha$ is the tolerance factor percentage greater than 0 and less than 100. We refer to the simulation phases in this fortunate situation as those in primary mode.

Otherwise, i.e., if the inequality (1) is not satisfied for some subdomain $n$ as shown in Figure 1, the simulation is performed in secondary mode. In this mode, every node, except for one node (12 in the figure), is responsible for a secondary subdomain having particles more than the average, in addition to its primary one. For example, the subdomain 22 has helper nodes 02, 30 and 33 shown in italic and blue letters in Figure 1. The particles in a densely populated subdomain are also distributed to its helper nodes as their secondary particles so that each node $n$ has $Q_n$ particles in total, which are the union of $Q_n^p$ primary particles in the primary subdomain $n$ and $Q_n^m$ secondary particles in the secondary subdomain $m$, satisfying the following inequality for balancing similar to (1) for all $n$.

$$Q_n = Q_n^p + Q_n^m \leq (P/N)(100 + \alpha)/100 = P_{\text{max}}$$

(2)

Note that since all but one nodes have secondary subdomains, a node whose primary subdomain is densely populated, e.g., node 22, is not only helped by other nodes but also helps another node 20, as the balancing algorithm discussed in Section 2.2 orders.

Also note that the load in secondary mode is balanced not only in the number of particles but also in the size of responsible subdomains, although the latter load is twice as heavy.

![Figure 1: Space domain partitioning.](image-url)
as that in the primary mode. This is another justification for making a node with densely populated primary subdomain help another node.

The examination whether the load is balanced well and the mode switching possibly with load rebalancing are performed as follows every simulation time step in which particles can move crossing subdomain boundaries.

1. If the inequality (1) is satisfied for all subdomains, the mode stays in or turns to primary. In the case of staying, only the particles crossing subdomain boundaries are transferred between nodes by neighboring communications. Otherwise, in addition to boundary crossing ones, particles that have been secondary are transferred to nodes responsible for them as primary particles.

2. If the current mode is secondary and the inequality (1) is not satisfied but (2) is satisfiable keeping the secondary subdomain assignment, the mode stays in secondary without global rebalancing. Particles may be transferred among the helpers and their helpand for the local load balancing in addition to the transfer of the particles crossing boundaries. The satisfiability check for (2) and the local balancing are discussed in Section 2.3.

3. Otherwise, the secondary subdomain assignments are performed (or modified) so that \( Q_n \) is equal to \( P/N \) for all \( n \) to accomplish perfect balancing. The subdomain assignment algorithm is discussed in Section 2.2.

### 2.2 Secondary Subdomain Assignment

When it is detected that the inequality (1) or (2) is unsatisfiable in primary or secondary mode respectively, secondary subdomains are assigned to nodes, by modifying the original assignment if the mode has already been in secondary, to accomplish perfect balancing.

The assignment algorithm is quite simple as follows.

(b1) Split the set of nodes into two disjoint subsets \( L = \{ n \mid P_n < P/N \} \) and \( G = \{ n \mid P_n \geq P/N \} \). Let the tentative value of \( Q_n \) be \( P_n \) for all \( n \).

(b2) Repeat the following steps (b3) through (b5) until \( L \) becomes empty.

(b3) Remove an element \( l \) from \( L \) such that \( Q_l = \min_{n \in L} \{Q_n\} \) and remove an element \( g \) from \( G \) as follows.

- If the mode is secondary and \( l \) has been helping a node \( n \) in \( G \), let \( g \) be \( n \).
- Otherwise, the node \( g \) is chosen such that \( Q_g = \max_{n \in G} \{Q_n\} \).

(b4) Assign the subdomain \( g \) to the node \( l \) as its secondary subdomain and also assign \( Q_l^f = (P/N) - Q_l \) particles in the subdomain \( g \) to the node \( l \) so that \( Q_l \leftarrow Q_l + Q_l^f = P/N \). Now \( Q_g \) becomes \( Q_g - Q_l^f \).

(b5) If \( Q_g < P/N \), add \( g \) to \( L \). Otherwise add \( g \) back to \( G \).

---

1 You may reduce the frequency of these operations by overlapping adjacent subdomains a little bit more heavily and by exploiting the fact that the velocity of a particle is limited to some upper bound, e.g., light speed.

2 We know English does not has such a word but dare to neologize to mean “the node helped by other nodes.”

3 If \( P \) is a multiple of \( N \). Otherwise, \( Q_n \) is \([P/N]\) or \([P/N]\), but we assume \( P \) is a multiple of \( N \) in this section for the sake of explanation simplicity.
(b6) If $G$ has two or more elements, pick an arbitrary element $r$ from $G$ and assign the subdomain $r$ to other nodes in $G$ without particle assignment. Otherwise, i.e., $G$ has only one element, let $r$ be this node.

It is obvious the algorithm stops making every node $n$ except for $r$ have a secondary subdomain and $Q_n = P/N$ for all $n$. As mentioned in Section 2.1, the key for perfect balancing is the step (b5) where we add $g$ with $P_g \geq P/N$ but $Q_g < P/N$ to $L$ so that it helps other node when it has deputed so many particles to its helpers that $Q_g$ becomes less than $P/N$ tentatively. Figure 2 shows an example balancing result for the particle distribution shown in Figure 1 providing we suddenly faces the imbalance due to, for example, initial particle positioning. The number of particles in each subdomain (a) and that assigned to each node (b) are illustrated by the bar whose color and numbers above and below it represent the subdomain and the node.

2.3 Checking and Keeping Local Balancing

In the secondary mode, the particle movements crossing subdomain boundaries could break the satisfiability of the inequality (2) if we stuck to the secondary subdomain assignment. To examine the satisfiability and to keep the local balancing among a helpand-helper family, we form a tree $T$ whose vertices are the computation nodes and edges represent helpand-helper relationship. That is, the root of the tree is the node $r$ defined in the step (b6) of the previous section, and the parent of a non-root node is its helpand. The tree corresponding to the balancing result in Figure 2(b) is show in Figure 3.

The examination of the satisfiability of (2) is performed by traversing the tree $T$ in a bottom-up (leaf-to-root) manner as follows.

(e1) Let a set of nodes $S$ be that of leaves of the tree $T$. Let $P_{n}^\text{min}$ be $P_n$ for all $n \in S$. If
there is an element \( n \in S \) such that \( P_n = P_{\text{min}}^n > P_{\text{max}} \), the examination fails.

(e2) Repeat the following steps (e3) and (e4) until \( S \) becomes \{\( r \}\).

(e3) Find a node \( n \) such that the set of its helpers \( H(n) \) is a subset of \( S \), and remove \( H(n) \) from \( S \).

(e4) Add \( n \) to \( S \) and let \( P_{\text{min}}^n \) be as follows.

\[
P_{\text{min}}^n = \max(0, P_n - \sum_{m \in H(n)} (P_{\text{max}} - P_{\text{min}}^m))
\]

If \( P_{\text{min}}^n > P_{\text{max}} \), the examination fails.

Since a leaf node does not have helpers, the failure in the step (e1) obviously means that the inequality (2) cannot be satisfied. As for the failure in (e4), since \( \sum_{m \in H(n)} (P_{\text{max}} - P_{\text{min}}^m) \) means the maximum particle amount which \( n \)'s helpers accommodate as their secondary particles and thus \( P_{\text{min}}^n \) is the minimum number of particles in \( n \) which the node \( n \) has to be responsible for, \( P_{\text{min}}^n > P_{\text{max}} \) leads us that the inequality (2) is unsatisfiable. Therefore, the algorithm is complete. On the other hand, when the algorithm stops at (e2) with \( P_{\text{min}}^n \leq P_{\text{max}} \) for all \( n \), it is assured that, for all \( n \), \( P_n \) particles can be distributed among \( n \) and its helpers keeping \( Q_m \leq P_{\text{max}} \) for all \( m \in F(n) \) where \( F(n) \) is defined as \( \{n\} \cup H(n) \). That is, even if \( n \) has to accommodate \( P_{\text{max}} - P_{\text{min}}^n \) particles for its helpand, \( P_n - P_{\text{min}}^n \) particles can be accommodated by its helpers because they are at most \( \sum_{m \in H(n)} (P_{\text{max}} - P_{\text{min}}^m) \).

Therefore, the algorithm is sound.

If the examination passes, a part of particles in a subdomain \( n \) are redistributed to the members of the family \( F(n) \), i.e., the node \( n \) and its helpers in \( H(n) \). The target of the redistribution is the following, where \( Q_k^n \) is the number of particles in the subdomain \( n \) and currently accommodated by the node \( k \).

- Particles currently in a node \( m \notin F(n) \), which has just crossed a boundary and moved into the subdomain \( n \) from other subdomain.

---

Figure 3: Helpand-helper tree for balancing result in Figure 2(b).
• Particles overflowed from a node $m \in F(n)$. More specifically, particles are overflowed from $m$ in either of the following cases.

- $m \neq n$ and $Q_m^n + P_{m\text{in}}^m > P_{\text{max}}$ and thus $Q_m^n + P_{m\text{in}}^m - P_{\text{max}}$ particles are overflowed to satisfy the minimum requirement defined by $P_{m\text{in}}^m$.

- $m = n$ and $Q_n^n + R_n > P_{\text{max}}$ where $R_n$ is the number of particles assigned to $n$ as the result of the redistribution for the family rooted by $p = \text{parent}(n)$ to which $n$ belongs as a helper. That is, $R_n = Q_p^n$ at the beginning of the next simulation step. The number of overflowed particles is $Q_n^n + R_n - P_{\text{max}}$.

Note that the criteria above are to minimize the amount of particle transfer rather than to minimize the load deviation among the nodes. Let $\Phi_n$ be the total number of redistributed particles defined above or, more specifically, be as follows.

$$\Phi_n = \sum_{k \notin F(n)} Q_k^n + \sum_{m \in H(n)} \max(0, Q_m^n + P_{m\text{in}}^m - P_{\text{max}}) + \max(0, Q_n^n + R_n - P_{\text{max}})$$

The local balancing in a helpand-helper family is partly achieved by the following algorithm traversing the tree $T$ in a top-down manner.

(d1) Let a set of node $S = \{r\}$, and $R_r = 0$.

(d2) Repeat the following steps (d3) to (d6) until $S$ becomes empty.

(d3) Remove a node $n$ from $S$. If $n$ is the leaf node, let $Q_n$ be $P_n + R_n$ and skip the following steps (d4) to (d6). Otherwise, add the helpers of $n$, i.e., $H(n)$, to $S$.

(d4) If the following inequality is satisfied:

$$P_n + R_n + \sum_{m \in H(n)} \max(P_{m\text{in}}^m, Q_m^m) \leq P_{\text{max}} \cdot |F(n)|$$

we need not to push down primary particles of any node $m$ to its own helpers. If this holds, let $B_m = \min(P_{\text{max}}, Q_m^n + \max(P_{m\text{in}}^m, Q_m^m))$ for all $m \in H(n)$ to represent the baseline number of particles above which we place particles to be redistributed as shown in Figure 4(a). Otherwise, let the baseline $B_m$ be $\min(P_{\text{max}}, Q_n^n + P_{m\text{in}}^m)$ to allow us to push down $Q_m^m - P_{m\text{in}}^m$ particles as shown in Figure 4(b). In both cases, let $B_n$, the baseline of $n$, be $\min(P_{\text{max}}, Q_n^n + R_n)$.

(a) without pushing down  
(b) with pushing down

Figure 4: Particle redistribution in a family.
(d5) Find the minimum subset $F'(n)$ of $F(n)$ such that the followings are satisfied.

\[
\forall m' \in F'(n), \forall m \in F(n) \cap F'(n) : B_{m'} \leq B_m \\
\forall m \in F(n) \cap \overline{F'(n)} : \Phi_n + \sum_{m' \in F'(n)} B_{m'} \leq B_m \cdot |F'(n)|
\]

(d6) Let $R_m$ for all $m \in H(n)$ and $Q_n$ be the followings.

\[
R'_m = \begin{cases} 
(\Phi_n + \sum_{m' \in F'(n)} B_{m'})/|F'(n)| - B_m & m \in F'(n) \\
0 & m \notin F'(n)
\end{cases}
\]

\[
R_m = R'_m \\
Q_n = B_n + R'_n
\]

The step (d5) is to find the leftmost three bars (nodes) in Figure 4(a) and (b) for the local load balancing among these lightly loaded nodes by distributing $R'_m$ given in the step (d6).
3 OhHelp Library

3.1 Library Layers

The OhHelp library package has three layers which are referred to as level-1, level-2 and level-3. The functions provided by each layer are summarized as follows.

**level-1:** This level provides a load-balancer function named `oh1_transbound()` which examines whether particles are distributed among nodes in a well-balanced manner, (re)builds helpand-helper configuration if necessary, and tells you how to move particles among nodes. That is, this function implements the OhHelp algorithm described in Section 2. In addition, level-1 library has functions for collective communications in helpand-helper families, and those for statistics and verbose messaging. See Section 3.4 for functions excluding those for statistics and verbose messaging which are explained in Section 3.8 and Section 3.9 respectively.

**level-2:** In this level, the load-balancer function `oh2_transbound()` does what its level-1 counterpart does, and transfers particles among nodes according to the schedule determined by the level-1 function. See Section 3.5 for detailed explanation of level-2 API functions.

**level-3:** Functions for particle manipulation added in this level are to determine the identifier of the subdomain where given particle resides. The other useful functions are for inter-node communications of arrays having vectors/scalars associated with grid points in a subdomain, i.e., those for electromagnetic field, current density, and so on. See Section 3.6 for detailed explanation of level-3 API functions.

Functions in each layer are composed in a level-specific source file, namely `ohhelp1.c`, `ohhelp2.c` and `ohhelp3.c` which require header files of same names, i.e., `ohhelp1.h`, `ohhelp2.h` and `ohhelp3.h`. To have a library of level-2 or level-3, it is required to compile lower level libraries as well, and thus you will have all functions in all layers if you are to use level-3 library. However, this does not mean that you have to use all functionalities provided by all level libraries. In fact, except for the essential functionality given by `oh1_transbound()`, you are almost free to pick functions you like to use. Therefore, API functions are named with prefixes `oh1_`, `oh2_` or `oh3_` to show which level they belong to.

This naming rule, however, could be too rigid for you to use all functions provided by your preferred layer and lower, because it will be tiresome to remember the layer number which a function belongs to. Therefore, the library has special header files `ohhelp_f.h` for Fortran programmers and `ohhelp_c.h` for those who love C, in order to give API functions aliases which just have a common prefix `oh_` as discussed in Section 3.10.

3.2 Applying OhHelp to PIC Simulators

Figure 5 shows a typical configuration of OhHelp'ed PIC simulators. In the figure, it is assumed that the baseline simulator to apply OhHelp is domain-decomposed and its main loop consists of four phases, `particle pushing`, `particle transferring`, `current scattering`, and `field solving` as follows.

**particle pushing:** Each node accelerates particles residing in the subdomain assigned to the node by electric and the Lorentz force law referring to electromagnetic field data \( E \) and \( B \) associated to the grid points in its subdomain. Then the node moves particles according to their updated velocities. Particle movements crossing subdomain boundaries will be taken care of by the next phase.
Figure 5: Typical 3D PIC simulator with OhHelp.

**particle transferring:** Each node transfers particles, which has crossed its subdomain boundaries, to the nodes responsible for adjacent subdomains.

**current scattering:** Each node calculates the contributions of the movement of its particles to the current density $J$ at the grid points in its subdomain. Then the boundary values of $J$ are exchanged between adjacent subdomains.

**field solving:** Each node locally updates the values of $E$ and $B$ at the grid points in its subdomain using, for example, leapfrog method to solve Maxwell’s equations. Then the boundary values of $E$ and $B$ are exchanged between adjacent subdomains.

Applying OhHelp to the baseline simulator outlined above is fairly easy. In fact, required modifications to the main simulation loop of the baseline simulator are just as follows.

**duplication of data structures:** Data structures for the subdomain and particles in it should be duplicated so that a node has primary and secondary subdomains and particles.

**duplication of computation:** The phases except for particle transferring of the main loop should be duplicated to locally update particle and field data.

**addition of collective communications:** Current densities for a secondary subdomain is calculated locally and thus should be summed up to have the complete data for the subdomain. The boundary or whole values of electromagnetic field should be broadcasted from each helpand to its helpers.

**attachment of load balancer:** To transfer particles among nodes, the library function for load balancing should be called to have the transfer schedule or to do the transfer itself.

In the following subsections, the modifications above are explained more detailedly.
3.2.1 Duplication of Data Structures

Since each node may have primary and secondary subdomains and particles, you have to duplicate data structure for electromagnetic field and current density to have those for primary subdomain and for secondary subdomain. For example, suppose the baseline simulator is coded in Fortran and the electromagnetic field for a subdomain is declared and allocated as;

```fortran
real*8, allocatable :: eb(:,:,:,:)  
allocate(eb(6, ϕ_lx:ϕ_ux−1, ϕ_lx:ϕ_uy−1, ϕ_lx:ϕ_uz−1))
```

where the first dimension is for three components of electric field vector and those of magnetic field vector, and ϕ_lx and ϕ_ux and their counterparts of y and z axes are lower and upper boundaries of the subdomain including a few planes for the overlap of adjacent subdomains. An OhHelp’ed version of this four-dimensional array has one additional dimension for primary and secondary ones and is declared and allocated as;

```fortran
real*8, allocatable :: eb(:,:,:,:,:)  
allocate(eb(6, ϕ_lx:ϕ_ux−1, ϕ_lx:ϕ_uy−1, ϕ_lx:ϕ_uz−1, 2))
```
to have primary field data in the subarray `eb(:,:,:,:,1)` while secondary data are stored in the other subarray `eb(:,:,:,:,2)`.

The other example for a C-coded simulator is given with the following declaration and allocation.

```c
struct ebfield {double ex,ey,ez,bx,by,bz} *eb;

eb = (struct ebfield*) malloc(sizeof(struct ebfield)*((ϕ_ux−ϕ_lx)((ϕ_uy−ϕ_ly)((ϕ_uz−ϕ_lz))));
```

A reasonable way to apply OhHelp to the example above is;

```c
struct ebfield {double ex,ey,ez,bx,by,bz} *eb[2];

eb[0] = (struct ebfield*) malloc(sizeof(struct ebfield)*((ϕ_ux−ϕ_lx)((ϕ_uy−ϕ_ly)((ϕ_uz−ϕ_lz))*2));

eb[1] = eb[0] + ((ϕ_ux−ϕ_lx)((ϕ_uy−ϕ_ly)((ϕ_uz−ϕ_lz)));
```

Note that, for both examples above, ϕ_lx, ϕ_ly and ϕ_lz in OhHelp’ed version could have to be larger than those in the original version because they must be for the largest subdomain in the system rather than for the primary subdomain for the node if the subdomain size is not uniform in the system. That is, the node should be able to be responsible for any subdomain in the system. Also note that it is not necessary to represent the electromagnetic field by one array, but you may have two arrays for electric and magnetic fields, or even six arrays for each component of electric and magnetic field vectors. However, you have to remember that splitting arrays should cost in the communication of them for boundary data exchange and broadcast and/or reduction in helpand-helper families.

On the other hand, adding a dimension to the array for particles to accommodate primary and secondary ones is not a good idea, because the number of particles in each category is not fixed. Therefore, the array must have \( P_{\text{max}} \) elements\(^4\) defined in the inequality (1) in Section 2.1. Then, in the node \( n \), the first part of the array should accommodate \( Q_n^p \) primary particles while the second part, which directly follows the first part, should have

\(^4\)If the total number of particles in the system fluctuates due to, for example, particle injection and/or removal, \( P \) for \( P_{\text{max}} \) calculation in the inequality (1) should be the maximum number of total particles in the simulation.
have $Q_p^n$ particles for $p = \text{parent}(n)$. The values of $Q_n^n$ and $Q_p^n$ are given by the library function for load balancing as discussed later.

The other remark on the array of particles is that if the array is partitioned into portions for $S$ species, the library should know it. For example, suppose the baseline simulator has two particle species, one for (super)ions and the other for (super)electrons, and the particle array is partitioned into two regions to store ions in the first region and electrons in the second region. This partitioning is done, for example, to save memory space eliminating species identifier and/or physical quantities of species such as the charge and mass of a particle from the array element representing a particle, and/or to save operations for the references to these quantities and for the calculations on them. Since the layout of two types of particles should be kept after the particle transfer, the library function for load balancing have to aware that $S = 2$ to make transfer schedule and, if you desire to do, to transfer particles. The function is also capable to report you the number of particles for each species and each of primary/secondary categories.

Note that particle transferring for a simulation step should consist of $S$ transfers for each species, a large $S$, say 10 or more, may cause a too large communication overhead to benefit from the array partitioning. Therefore, if your simulation has a large number of species, it is recommended to attach the species identifier and/or the physical quantities to each particle and tell the library that $S = 1$.

Also note that if you apply level-2 or level-3 library\(^5\), a particle should be represented by a structured data which should include particle position coordinates, velocity vector components, and other necessary information as discussed in Section 3.5.1. Otherwise, i.e., if you use level-1 only and transfer particles among nodes by yourself, the set of particles accommodated in a node can be represented in two or more arrays paying some communication overhead.

### 3.2.2 Duplication of Computation

Since a particle is accommodated by only one node, the node is of course fully responsible for the particle. Therefore, each node should perform particle pushing and current scattering for its primary and secondary particles. A reasonable way to implement this duplicated computation for particles is to call functions corresponding to the operations twice.

For example, if your simulator has a Fortran subroutine named `particle_push()` with three arguments for the particle array, its size and electromagnetic field, fundamental operation to duplicate particle pushing is easy as follows, providing the array `pbuf` has particles each of which is represented by a structured data.

```fortran
    call particle_push(pbuf(1), Q_n^n, eb(:,:,1,1))
    call particle_push(pbuf(Q_n^n+1), Q_p^n, eb(:,:,2,2))
```

However, this is not sufficient because two instances of `particle_push()` should have different base coordinates by which the particle position in the coordinate system of whole domain is mapped onto local coordinate system for a subdomain. That is, suppose the base simulator calculates the particle velocity in `particle_push()` by:

```fortran
    call lorentz(eb, pbuf(i)x-xl, pbuf(i)y-yl, pbuf(i)z-zl, acc(1:3))
    pbuf(i)x = pbuf(i)x + acc(1)
    pbuf(i)y = pbuf(i)y + acc(2)
    pbuf(i)z = pbuf(i)z + acc(3)
```

\(^5\)Unless you choose partial application of level-3 disabling level-2 functions, which is discussed in Section 3.6.2.
where the structure elements \(x, y, z, vx, vy\) and \(vz\) are for \(x/y/z\)-components of the position and the velocity of the \(i\)-th particle. \texttt{lorentz()} is the subroutine to calculate acceleration vector \(\texttt{acc(1:3)}\) referring to electromagnetic field vectors on the grid points surrounding the particle, and \(xl, yl\) and \(zl\) are the base coordinates of the subdomain, i.e., the coordinates of the west-south-bottom corner of the subdomain.

The code above should be modified to refer to subdomain dependent base coordinates. A reasonable way is to have a map of subdomain boundaries, say \(\texttt{sdoms(2,3,}N\texttt{)}\) whose element \(\texttt{sdoms(}\beta\texttt{,}d\texttt{,}n\texttt{)}\) has lower \((\beta = 1)\) or upper \((\beta = 2)\) boundary of \(d\)-th dimension of the subdomain \(n\). With this array, the modified version of \texttt{particle.push()} has an additional array argument, say \(\texttt{sdom(2,3)}\) for the subdomain in problem and is called as follows where \(p = \texttt{parent}(n)\).

\[
\begin{align*}
call\;\texttt{particle.push(pb自负(1), \ Q_{n_{自负}}, \ \texttt{eb(:, :, :, 1)}, \ \texttt{sdom(:, :, n)})} \\
call\;\texttt{particle.push(pb自负(Q_{n_{自负}}+1), \ Q_{n_{自负}}, \ \texttt{eb(:, :, :, 2)}, \ \texttt{sdom(:, :, p)})}
\end{align*}
\]

Then at the beginning of the body of \texttt{particle.push()}, the following assignment is added for the base coordinates where \(\texttt{sdom}\) is the fourth argument array passed to the subroutine.

\[
\begin{align*}
xl &= \texttt{sdom(1,1)} \\
yl &= \texttt{sdom(1,2)} \\
zl &= \texttt{sdom(1,3)}
\end{align*}
\]

Note that the upper boundaries \(\texttt{sdom(2,:)}\) will also be used in the function to detect the particles crossing the subdomain boundaries. Remember that you are responsible for counting number of particles in each subdomain, each species and each primary/secondary category and for reporting it to the library. Also note that the array equivalent to \(\texttt{sdoms(:, :, :)}\) can be given by the initialization function of level-3 library as discussed in Section 3.6.1. A C-code version of the example above looks as follows.

\[
\begin{align*}
\texttt{particle.push(pb自负, \ Q_{n_{自负}}, \ eb[0], \ sdom[n]);} \\
\texttt{particle.push(pb自负+Q_{n_{自负}}, \ Q_{n_{自负}}, \ eb[1], \ sdom[p]);}
\end{align*}
\]

The modification of current scattering can be implemented similarly, but it needs collective communications to sum local results of the scattering calculated by nodes in the family. The sum is obtained by a simple reduce operation or by an all-reduce operation to share the sum in family members, depending on the implementation of field solving as discussed below. Also, the (all-)reduce communication is discussed in Section 3.2.3.

As for field solving, there are two reasonable ways to modify its baseline implementation. The first candidate is to simply broadcast the solution of primary subdomain from each helpand to its helpers. That is, each node updates electromagnetic field vectors in its primary subdomain, exchanges boundary data between adjacent nodes, and broadcasts the
whole field vectors in its primary subdomain and a few boundary planes to its helpers by
the method discussed in Section 3.2.3. In this broadcast-type implementation, since the
current density on each grid point in a subdomain is referred to only by the node responsible
for the subdomain as primary, summing current densities will be performed by a simple
one-way reduction followed by boundary exchange.

The other candidate is to duplicate the calculation of field solving. That is, each node
updates electromagnetic field vectors in both primary and secondary subdomains. A rea-
sonable way to obtain boundary values is to exchange boundary planes of adjacent primary
subdomains and then to broadcast planes to the helpers. In this duplicate-type implement-
tation, since the current density on each grid point in a subdomain is referred to by all the
nodes responsible for the subdomain as primary or secondary, summing current densities
will be performed by an all-reduce communication followed by boundary exchange between
primary subdomains and broadcast boundary planes from the helpand to its helpers.

The choice from these two candidates should be determined by trading off the compu-
tation cost of field solving and the communication cost of broadcasting. In practice, if your
simulator performs one leapfrog solving per one simulation step, the duplicate-type should
be chosen because a leapfrog update of a subdomain is faster than broadcast. On the other
hand, if your simulator adopts sub-stepping method to iterate leapfrog multiple times in a
simulation step with, for example, particle-fluid hybrid method, the broadcast-type can be
better.

3.2.3 Addition of Collective Communications
As discussed in Section 3.2.2, you need to add at least the following collective communica-
tions.

- A simple one-way reduction or an all-reduce communication to sum the current density
  among family members. In the latter case, the current density vectors of grid points
  in boundary planes should be broadcasted from the helpand to its helpers.
- Broadcast of electromagnetic field vectors of the whole of or the boundaries of the
  subdomain from the helpand to its helpers.
- Broadcast of electromagnetic field vectors when the helpand-helper tree is reconfigured
  due to an unacceptable imbalance and each node has new helpand.

Fundamentally, the collective operations above are performed by MPI functions, MPI-
Reduce() or MPI_Allreduce() and MPI_Bcast() with argument comm being the commu-
icator for the family which each node belongs to. Simply calling these functions, however,
should cause a severe performance problem because a node may belong to two families, one
as a helpand and the other as a helper. That is, if we carelessly perform collective com-
munications by doing them, for example, as a helpand and then as a helper, it may cause
unnecessary serialization because the root family must wait the completion of the commu-
nications in the second generation families which must wait those in the third ones and
so on. Reversing the helpand/helper order cannot solve the problem because the bottom
families must wait the completion in the second-bottom ones and so on.

This problem is solved by a simple red-black technique which paints families of odd-
number generations by red and even ones by black and performs communications of red
families first and then of black families. Since families of same color are mutually exclusive,
the communications among them are performed in parallel.

The library provides various means for the red-black collective communications as fol-
lows.
• Level-1 library manages the family communicators and report you the communicators for the families which the local node belongs to, together with their colors and the ranks of the roots in the communicators. These information is sufficient to implement your own version of collective communications besides those provided by the library shown below.

• Level-1 library also provides you with functions for one-way reduction, all-reduce and broadcast with given data buffers, data counts and data types. All of these functions take care of the red-black ordering and special treatment for the tree root and leaves, each of which belongs to only one family.

• Level-3 library provides functions for you to perform one-way reduction, all-reduce and broadcast of the current vectors, electromagnetic field, and other arrays, for example that having charge densities, if necessary. The usage of the functions is much simpler than the level-1 counterparts, because you simply need to register each of field-arrays, which are arrays for current density vectors, electromagnetic field and so on having elements associated to the grid points in a subdomain, and call these functions with primary and secondary arrays and the identifier of the array.

• Level-3 library also provides a function to exchange boundaries of field-arrays optionally followed by broadcast of boundary data from the helpand to its helpers.

3.2.4 Attachment of Load Balancer

Attaching OhHelp load balancer to your simulator is of course essential. What you need to do is simply calling oh1_transbound(), where l is level number in \{1, 2, 3\} with a few explicit arguments and gives it a histogram of particles accommodated by the local node. That is, if your code is written in Fortran, you have to have an array, say npgram(N, S, 2) whose element npgram(m+1, s, c) has the number of particles residing in the subdomain m ∈ [0, N], categorized in the species s ∈ [1, S] and accommodated by the local node as its primary (c = 1) or secondary (c = 2) ones.

Similarly, C-coded simulator should have a conceptually three-dimensional array npgram[N × S × 2] whose element npgram[m + N(s + Sc)] has the number of particles residing in the subdomain m, categorized in the species s ∈ [0, S − 1] and accommodated by the local node as its primary (c = 0) or secondary (c = 1) ones. If you like to access the array element by npgram[c][s][m], you have to do the followings.

```
int **npgram[2];
nphgram[0] = (int**)malloc(sizeof(int*)*S*2);
nphgram[1] = npgram[0] + S;
nphgram[0][0] = (int*)malloc(sizeof(int)*N*S*2);
nphgram[1][0] = npgram[0][0] + N*S;
for (i=0; i<S; i++) for (j=1; j<S; j++)
    npgram[i][j] = npgram[i][j-1] + N;
```

The main output you will obtain from the level-1 oh1_transbound() is a pair of (conceptually) three-dimensional arrays, say rcounts(N, S, 2) and scounts(N, S, 2) for Fortran or rcounts[N × S × 2] and scounts[N × S × 2] for C, which tell you incoming and outgoing particle transfer schedules. That is, rcounts(m+1, s, c) and scounts(m+1, s, c) notify you how many particles of species s should be received/sent from/to the node m as receiver’s primary (c = 1) or secondary (c = 2) ones. Similary, rcounts[m + N(s + Sc)]
and \texttt{scounts} \[ m + N(s + Sc) \] tell you the receiving/sending counts of primary \((c = 0)\) or secondary\((c = 1)\) particles for the node \(m\) and species \(s\).

On the other hand, level-2 function \texttt{oh2_transbound()} and its level-3 counterpart \texttt{oh3_transbound()} perform particle transfer on behalf of you. To make the functions do the job easily, you have to give an additional tip to show which subdomain each particle has moved. That is, each element of the array of particles, say \texttt{pbuf}, should be a structured data having an element \texttt{nid} to have the identifier of the subdomain where the particle is residing after particle pushing. Therefore, your subroutine/function for particle pushing should modify this element for each particle which has just crossed the subdomain boundary. Remember that level-3 library has functions to calculate the subdomain identifier from the particle position.

An important notice is that the transfer schedule given by \texttt{oh1_transbound()} and that used in \texttt{oh2_transbound()} and \texttt{oh3_transbound()} are unaware of particle positions. That is, in secondary mode, a pair of closely located particles may be parted from each other to be accommodated by two different nodes in the family for the subdomain where the pair resides. Therefore, if your simulator takes care of proximal particle-particle interactions using, for example, Monte Carlo Collision method, you have to wait until the OhHelp library evolves to be position-aware.

### 3.3 Dimension of Simulated Space

The OhHelp library can be applied to PIC simulations of one-dimensional, two-dimensional or three-dimensional space domain. For the sake of efficiency, however, the number of dimensions \(D\) is hard-coded in the library source code using a C constant macro named \texttt{OH\_DIMENSION} whose default value is three. Therefore, if your simulator is one- or two-dimensional, you have to explicitly define the macro through the compiler option \texttt{-DOH\_DIMENSION=1} or \texttt{-DOH\_DIMENSION=2}, or have to edit the header file \texttt{oh\_dim.h} in which the default definition of \texttt{OH\_DIMENSION} is given as follows.

```c
#ifndef OH_DIMENSION
#define OH_DIMENSION 3
#endif
```

Remember that \texttt{oh\_dim.h} is included by \texttt{ohhelp\_f.h} and \texttt{ohhellp\_c.h} for function aliasing and thus modifying \texttt{oh\_dim.h} is easier to have consistent definition if you use aliases.

### 3.4 Level-1 Library Functions

Level-1 library provides the following functions.

\texttt{oh1_init()} receives fundamental parameters and arrays by which the library interacts with your simulator body, and initializes internal data structures.

\texttt{oh1_transbound()} implements the core algorithm of OhHelp and reports the particle transfer schedule.

\texttt{oh1_broadcast()} performs broadcast communication in helpand-helper families.

\texttt{oh1_all_reduce()} performs all-reduce communication in helpand-helper families.

\texttt{oh1_reduce()} performs simple one-way reduce communication in helpand-helper families.

\texttt{oh1_init_stats()}\n\texttt{oh1_stats_time()}
oh1_show_stats()
oh1_print_stats()

See Section 3.8 for the functions for statistics above.

oh1_verbose() is explained in Section 3.9.

The function API for Fortran programs is given by the module named ohhelp1 in the file oh_mod1.F90, while API for C is embedded in ohhelp_c.h.

3.4.1 oh1_init()

The function (subroutine) oh1_init() receives a few fundamental parameters and arrays through which oh1_transbound() interacts with your simulator body. It also initializes internal data structures used in level-1 library. Among its thirteen arguments, other library functions directly refer to only the contents of the argument array nphgram as their implicit inputs. Therefore, after the call of oh1_init(), modifying the bodies of other arguments has no effect to library functions.

Fortran Interface

subroutine oh1_init(sdid, nspec, maxfrac, nphgram, totalp, rcounts, &
                         scounts, mycomm, nbor, pcoord, stats, repiter, verbose)
  use oh_type
  implicit none
  integer,intent(out) :: sdid(2)
  integer,intent(in) :: nspec
  integer,intent(in) :: maxfrac
  integer,intent(inout) :: nphgram(:,:,:)
  integer,intent(out) :: totalp(:,:)
  integer,intent(out) :: rcounts(:,:,:)
  integer,intent(out) :: scounts(:,:,:)
  type(oh_mycomm),intent(out) :: mycomm
  integer,intent(inout) :: nbor(3,3,3) ! for 3D codes.
  integer,intent(in) :: pcoord(OH_DIMENSION)
  integer,intent(in) :: stats
  integer,intent(in) :: repiter
  integer,intent(in) :: verbose
end subroutine

sdid(2) will have the identifiers of primary and secondary subdomain of the local node in sdid(1) and sdid(2) respectively. Therefore, sdid(1) is always equivalent to the MPI rank number of the calling process. On the other hand, sdid(2) initially has −1 to mean we are in primary mode initially, but will be set to a non-negative number in \([0,N−1]\) to identify the secondary subdomain by oh1_transbound() if it turns the mode to secondary. Note that, even in secondary mode, sdid(2) may have −1 if the local node is the root of the helpand-helper tree.

nspec should have the number of species \(S\).

maxfrac should have the tolerance factor percentage of load imbalance \(\alpha\) greater than 0 and less than 100.

nphgram\((N,S,2)\) should be an array whose element \(nphgram(m+1,s,c)\) should have the number of particles residing in the subdomain \(m \in [0,N−1]\) categorized in the species
s ∈ [1, S] and accommodated by the local node as its primary (c = 1) or secondary (c = 2) ones. The contents of the array can be undefined at the call of oh1_init() but must be completely defined at the call of oh1_transbound(). Upon returning from oh1_init() and oh1_transbound(), the contents of the array will be zero-cleared, so that you can (re)start counting.

**totalp(S, 2)** should be an array whose element totalp(s, c) will have the number of primary (c = 1) or secondary (c = 2) particles of species s to be accommodated by the local node as the result of load balancing performed by oh1_transbound(). Note that oh1_init() does not give any values to the array.

**rcounts(N, S, 2)** should be an array whose element rcounts(m+1, s, c) will have the number of particles of species s which the local node should receive from the node m as primary (c = 1) or secondary (c = 2) ones of the local node, after each call of oh1_transbound(). Remember that rcounts(n+1, s, c) for the local node n itself can be non-zero when it has particles residing in its primary (secondary) subdomain moving to its secondary (primary) subdomain.

**scounts(N, S, 2)** should be an array whose element scounts(m+1, s, c) will have the number of particles of species s which the local node should send to the node m as primary (c = 1) or secondary (c = 2) ones of the node m (not of the local node), after each call of oh1_transbound(). Remember that scounts(n+1, s, c) for the local node n itself can be non-zero when it has particles residing in its primary (secondary) subdomain moving to its secondary (primary) subdomain.

**mycomm** should be a structured data of oh_mycomm type whose definition is given in oh_type.F90 and will have the following integers when oh1_transbound() (re)builds a new helpand-helper configuration.

- **prime** is the MPI communicator for the family which the local node belongs to as the helpand, or MPI_COMM_NULL if it is a leaf of the helpand-helper tree.
- **sec** is the MPI communicator for the family which the local node belongs to as a helper, or MPI_COMM_NULL if it is the root of the helpand-helper tree.
- **rank** is the rank of the local node in the prime communicator, or −1 if it is a leaf.
- **root** is the rank of the helpand node in the sec communicator, or −1 if it is the root.
- **black** is 0 if the prime communicator is colored red, or 1 if colored black.

That is, **oh_mycomm** is defined as follows.

```f90
  type oh_mycomm
  sequence
    integer :: prime, sec, rank, root, black
  end type oh_mycomm
```

**nbor(3, . . . , 3)** should be a D-dimensional array of three elements for each dimension and its element nbor(ν1, . . . , νD) (νd ∈ [1, 2, 3]) must have the identifier of the subdomain adjacent to the primary subdomain of the local node. More specifically, let (π1, . . . , πD) be the coordinates for the local node in a conceptual D-dimensional integer coordinate system in which computational nodes (or equivalently their primary subdomains) are laid out, and rank(π1', . . . , πD') be the function to map the grid point
\[
\begin{align*}
\text{nbor(1,3)} &= \text{(*nbor)(2)[0]} \\
&= \text{rank}(v_p - 1, v_q + 1) \\
&= (v_p - 1) \times H_y + (v_q + 1) \\
&= n - 1 + H_y \\
\text{nbor(2,3)} &= \text{(*nbor)(2)[1]} \\
&= \text{rank}(v_p, v_q + 1) \\
&= v_p + H_y (v_q + 1) \\
&= n + H_y \\
\text{nbor(3,3)} &= \text{(*nbor)(2)[2]} \\
&= \text{rank}(v_p + 1, v_q + 1) \\
&= (v_p + 1) \times H_y (v_q + 1) \\
&= n + 1 + H_y \\
\text{nbor(1,2)} &= \text{(*nbor)(1)[0]} \\
&= \text{rank}(v_p - 1, v_q) \\
&= (v_p - 1) \times H_y v_q \\
&= n - 1 \\
\text{nbor(2,2)} &= \text{(*nbor)(1)[1]} \\
&= \text{rank}(v_p, v_q) \\
&= v_p + H_y v_q \\
&= n \\
\text{nbor(3,2)} &= \text{(*nbor)(1)[2]} \\
&= \text{rank}(v_p + 1, v_q) \\
&= (v_p + 1) \times H_y v_q \\
&= n + 1 \\
\text{nbor(1,1)} &= \text{(*nbor)(0)[0]} \\
&= \text{rank}(v_p - 1, v_q - 1) \\
&= (v_p - 1) \times H_y (v_q - 1) \\
&= n - 1 - H_y \\
\text{nbor(2,1)} &= \text{(*nbor)(0)[1]} \\
&= \text{rank}(v_p, v_q - 1) \\
&= v_p + H_y (v_q - 1) \\
&= n - H_y \\
\text{nbor(3,1)} &= \text{(*nbor)(0)[2]} \\
&= \text{rank}(v_p + 1, v_q - 1) \\
&= (v_p + 1) \times H_y (v_q - 1) \\
&= n + 1 - H_y \\
\text{nbor(6,0)} &= \text{(*nbor)(6)[0]} \\
&= \text{rank}(v_p - 1, v_q) \\
&= (v_p - 1) \times H_y v_q \\
&= n - 1 \\
\text{nbor(5,0)} &= \text{(*nbor)(5)[0]} \\
&= \text{rank}(v_p - 2, v_q) \\
&= (v_p - 2) \times H_y v_q \\
&= n - 2 \\
\end{align*}
\]

Figure 6: \text{nbor} and its default setting in $H_x \times H_y$ node coordinate system given by \text{pcoord}.
\((\pi_1, \ldots, \pi_D)\) to the identifier (MPI rank) of the node located at the point. With these definitions, an element of the array \texttt{nbor} should have the following (Figure 6).

\[
\texttt{nbor}(\nu_1, \ldots, \nu_D) = \text{rank}(\pi_1 + \nu_1 - 2, \ldots, \pi_D + \nu_D - 2)
\]

If \(D = 3\), for example, \texttt{nbor(1,1,1)} should have the identifier of the node whose primary subdomain contacts with that of the local node only at its west-south-bottom corner. \texttt{nbor(1,2,3)} should be the east neighbor of the local node, and \texttt{nbor(2,2,2)} is the local node itself.

Note that the neighboring relationship may or may not be periodic along each axis. That is, if the node coordinate system is \([0, \Pi_x - 1] \times [0, \Pi_y - 1] \times [0, \Pi_z - 1]\) and the local node is located at \((0, 0, 0)\), it may have west neighbor \((\Pi_x - 1, 0, 0)\) while its south neighbor can be nonexistent. In the latter case for nonexistent neighbors, \texttt{nbor} can have elements being \(-2\) (or less) to indicate that the corresponding neighboring grid points have no nodes. Also note that nonexistent neighbors can be found not only outside the node coordinate system but also in its inside for, e.g., holes.

Alternatively, if the work to define \texttt{nbor} is tiresome for you, you may delegate it to \texttt{oh1_init()} by making \texttt{nbor(1, \ldots, 1)} = \(-1\), and giving the size of node coordinate system \(H_1 \times \cdots \times H_D = N\) through the argument array \texttt{pcoord(D)=(/H_1, \ldots, H_D/)}.

In this case, \texttt{oh1_init()} initializes \texttt{nbor} assuming fully periodic coordinate system of \([0, H_1 - 1] \times \cdots [0, H_D - 1]\) and \(r = \text{rank}(\pi_1, \ldots, \pi_D)\) is given as follows.

\[
\begin{align*}
\ r_D &= \pi_D \\
\ r_d &= \pi_d + 1 \\
\ r &= r_1
\end{align*}
\]

\texttt{pcoord(D)} should be an array whose element \texttt{pcoord(d)} has the size of the \(d\)-th dimension \(H_d\) of the conceptual integer coordinate system of \([0, H_1 - 1] \times \cdots \times [0, H_D - 1]\) in which \(N = H_1 \times \cdots \times H_D\) computational nodes are layed out, if you delegate the setting of the array \texttt{nbor(3, \ldots, 3)} to \texttt{oh1_init()}. Otherwise, the array can have any values.

\texttt{stats} defines how statistics data is collected. See Section 3.8 for more details.

\texttt{repiter} defines how frequent statistics data is reported when \texttt{stats} = 2. See Section 3.8 for more details.

\texttt{verbose} defines how verbosely the execution progress is reported. See Section 3.9 for more details.

C Interface

```c
void oh1_init(int **sdid, int nspec, int maxfrac, int **nphgram,
              int **totalp, int **rcounts, int **scounts, void *mycomm,
              int **nbor, int *pcoord, int stats, int repiter, int verbose);
```

\**sdid** should be a double pointer to an array of two elements, or a pointer to NULL (not NULL itself) to order \texttt{oh1_init()} to allocate the array and return the pointer to it through the argument. The array will have the identifiers of primary and secondary subdomain of the local node in \((\ast\texttt{sdid})[0]\) and \((\ast\texttt{sdid})[1]\) respectively. Therefore, \((\ast\texttt{sdid})[0]\) is always equivalent to the MPI rank number of the calling process. On the other hand, \((\ast\texttt{sdid})[1]\) initially has \(-1\) to mean we are in primary mode.
initially, but will be set to a non-negative number in $[0, N-1]$ to identify the secondary subdomain by \texttt{oh1_transbound()} if it turns the mode to secondary. Note that, even in secondary mode, \((\ast \texttt{sdid})[1]\) may have $-1$ if the local node is the root of the helpand-helper tree.

\texttt{nspec} should have the number of species $S$.

\texttt{maxfrac} should have the tolerance factor percentage of load imbalance $\alpha$ greater than 0 and less than 100.

\texttt{**nphgram} should be a double pointer to an array of $2 \times S \times N$ elements to form \texttt{nphgram}[2][S][N] conceptually, or a pointer to NULL (not NULL itself) to order \texttt{oh1_init()} to allocate the array and return the pointer to it through the argument. Its element \texttt{nphgram}[c][s][m]\footnote{For the sake of conciseness, an element of conceptual $n$-dimensional array $a$ of $m_0 \times \cdots \times m_{n-1}$ elements, which is one-dimensional in reality, is denoted by $a[i_0] \ldots [i_{n-1}]$ which should be $a[j]$ in reality where $j$ is defined by $j_0 = i_0, j_k = i_k + j_{k-1}m_k, j = j_{n-1}$. Therefore \texttt{nphgram}[c][s][m] is \((\ast \texttt{nphgram})[m + N(s + Sc)]\) in reality.} has the number of particles residing in the subdomain $m \in [0, N-1]$, categorized in the species $s \in [0, S-1]$ and accommodated by the local node as its primary ($c = 0$) or secondary ($c = 1$) ones. The contents of the array can be undefined at the call of \texttt{oh1_init()} but must be completely defined at the call of \texttt{oh1_transbound().} Upon returning from \texttt{oh1_init()} and \texttt{oh1_transbound()}, the contents of the array will be zero-cleared, so that you can (re)start counting.

\texttt{**totalp} should be a double pointer to an array of $2 \times S$ elements to form \texttt{totalp}[2][S] conceptually, or a pointer to NULL (not NULL itself) to order \texttt{oh1_init()} to allocate the array and return the pointer to it through the argument. Its element \texttt{totalp}[c][s] will have the number of primary ($c = 0$) or secondary ($c = 1$) particles of species $s$ to be accommodated by the local node as the result of load balancing performed by \texttt{oh1_transbound()}. Note that \texttt{oh1_init()} does not give any values to the array.

\texttt{**rcounts} should be a double pointer to an array of $2 \times S \times N$ elements to form \texttt{rcounts}[2][S][N] conceptually, or a pointer to NULL (not NULL itself) to order \texttt{oh1_init()} to allocate the array and return the pointer to it through the argument. Its element \texttt{rcounts}[c][s][m] will have the number of particles of species $s$ which the local node should receive from the node $m$ as primary ($c = 0$) or secondary ($c = 1$) ones of the local node, after each call of \texttt{oh1_transbound().} Remember that \texttt{rcounts}[c][s][n] for the local node $n$ itself can be non-zero when it has particles residing in its primary (secondary) subdomain moving to its secondary (primary) subdomain.

\texttt{**scounts} should be a double pointer to an array of $2 \times S \times N$ elements to form \texttt{scounts}[2][S][N] conceptually, or a pointer to NULL (not NULL itself) to order \texttt{oh1_init()} to allocate the array and return the pointer to it through the argument. Its element \texttt{scounts}[c][s][m] will have the number of particles of species $s$ which the local node should send to the node $m$ as primary ($c = 0$) or secondary ($c = 1$) ones of the node $m$ (not of the local node), after each call of \texttt{oh1_transbound().} Remember that \texttt{scounts}[c][s][n] for the local node $n$ itself can be non-zero when it has particles residing in its primary (secondary) subdomain moving to its secondary (primary) subdomain.

\texttt{*mycomm} should be a pointer to a structured data named \texttt{S\_mycomm} whose definition is given in \texttt{ohhelp\_c.h}. Alternatively, it can be NULL (itself) if you do not want to
bother to play with family communicators but use only library functions for collective communications among family members. If you give the pointer to a \texttt{S\_mycommc} structure, you will have the followings when \texttt{ohi\_transbound()} (re)builds a new helpand-helper configuration.

\textbf{MPI\_comm prime} is the MPI communicator for the family which the local node belongs to as the helpand, or \texttt{MPI\_COMM\_NULL} if it is a leaf of the helpand-helper tree.

\textbf{MPI\_comm sec} is the MPI communicator for the family which the local node belongs to as a helper, or \texttt{MPI\_COMM\_NULL} if it is the root of the helpand-helper tree.

\textbf{rank} is the rank of the local node in the \texttt{prime} communicator, or \texttt{-1} if it is a leaf.

\textbf{root} is the rank of the helpand node in the \texttt{sec} communicator, or \texttt{-1} if it is the root.

\textbf{int black} is 0 if the \texttt{prime} communicator is colored red, or 1 if colored black.

That is, \texttt{S\_mycommc} is defined as follows.

\begin{verbatim}
struct S_mycommc {
    MPI_Comm prime, sec;
    int rank, root, black;
};
\end{verbatim}

**nbor** should be a double pointer to an array of $3^D$ elements to form \texttt{nbor}[3]...[3] conceptually, or a pointer to \texttt{NULL} (not \texttt{NULL} itself) if you want the library to allocate and initialize the array and return the pointer to it through the argument. If you prepare the array, its element \texttt{nbor}[\nu_D-1]...[\nu_0] ($\nu_d \in [0,1,2]$) must have the identifier of the subdomain adjacent to the primary subdomain of the local node. More specifically, let $(\pi_0, \ldots, \pi_{D-1})$ be the coordinates for the local node in a conceptual $D$-dimensional integer coordinate system in which computational nodes (or equivalently their primary subdomains) are laid out, and \texttt{rank}(\pi_0, \ldots, \pi_{D-1}) be the function to map the grid point $(\pi_0, \ldots, \pi_{D-1})$ to the identifier (MPI rank) of the node located at the point. With these definitions, an element of the array \texttt{nbor} should have the following (Figure 6).

\[
\texttt{nbor}[\nu_D-1]...[\nu_0] = \texttt{rank}(\pi_0 + \nu_0 - 1, \ldots, \pi_{D-1} + \nu_{D-1} - 1)
\]

If $D = 3$, for example, \texttt{nbor}[0][0][0] should have the identifer of the node whose primary subdomain contacts with that of the local node only at its west-south-bottom corner, \texttt{nbor}[2][1][0] should be for the node which shares west-top edge of the local node, \texttt{nbor}[1][1][2] should be the east neighbor of the local node, and \texttt{nbor}[2][2][2] is the local node itself.

Note that the neighboring relationship may or may not be periodic along each axis. That is, if the node coordinate system is $[0, \Pi_{z-1}] \times [0, \Pi_{y-1}] \times [0, \Pi_{x-1}]$ and the local node is located at $(0,0,0)$, it may have west neighbor $(\Pi_{x-1},0,0)$ while its south neighbor can be nonexistent. In the latter case for nonexistent neighbors, \texttt{nbor} can have elements being $-2$ (or less) to indicate that the corresponding neighboring grid points have no nodes. Also note that nonexistent neighbors can be found not only outside the node coordinate system but also in its \textit{inside} for, e.g., \textit{holes}.

Alternatively, if the work to define \texttt{nbor} is tiresome for you, you may delegate it to \texttt{ohi\_init()} by passing a pointer to \texttt{NULL} or by making **\texttt{nbor} = \texttt{-1}, and giving the size of node coordinate system \texttt{$H_0 \times \cdots \times H_{D-1} = N$} through the argument array \texttt{pcoord[D]} = $\{H_0, \ldots, H_{D-1}\}$. In this case, \texttt{ohi\_init()} initializes
(\*nbor) assuming fully periodic coordinate system of \([0, II_1-1] \times \ldots [0, II_D-1]\) and \(r = rank(\pi_0, \ldots, \pi_{D-1})\) is given as follows.

\[ r_{D-1} = \pi_{D-1} \quad r_d = r_{d+1} II_d + \pi_d \quad r = r_0 \]

\(\*pcoord\) should be a pointer to an array of \(D\) elements and each element \(pcoord[d]\) should have the size of the \(d\)-th dimension \(\Pi_d\) of the conceptual integer coordinate system of \([0, II_0-1] \times \cdots \times [0, II_D-1]\) in which \(N = II_0 \times \cdots \times II_D-1\) computational nodes are layed out, if you delegate the setting of the array (\*nbor)\([3^D]\) to oh1_init(). Otherwise, \(pcoord\) can be NULL or the array can have any values.

\(\text{stats}\) defines how statistics data is collected. See Section 3.8 for more details.

\(\text{repiter}\) defines how frequent statistics data is reported when \(\text{stats} = 2\). See Section 3.8 for more details.

\(\text{verbose}\) defines how verbosely the execution progress is reported. See Section 3.9 for more details.

3.4.2 oh1_transbound()

The function oh1_transbound() performs global collective communications of \(\text{nphgram}\) to examines whether the number of particles in nodes are well balanced. If it finds load imbalance is unacceptably large, it (re)builds helpand-helper configuration updating the structure \(\text{mycomm}\). Finally, it makes particle transfer schedule to report it through the array \(\text{rcounts} \) and \(\text{scounts}\), and updates the array \(\text{totalp}\) so that the array has the number of particles accommodated by the local node after the particle transfer. It also makes \(\text{nphgram}\) zero-cleared to give initial values of particle counting in the next simulation step. Note that the arrays \(\text{nphgram}, \text{rcounts}, \text{scounts}\) and \(\text{totalp}\) and the structure \(\text{mycomm}\) were given to oh1_init() as its arguments.

Besides these global arrays and structure, oh1_transbound() takes two arguments and returns an integer value to show you the mode in the next simulation step, as follows.

\textbf{Fortran Interface}

\begin{verbatim}
integer function oh1_transbound(currmode, stats)
  implicit none
  integer,intent(in) :: currmode
  integer,intent(in) :: stats
end function
\end{verbatim}

\textbf{C Interface}

\begin{verbatim}
int oh1_transbound(int currmode, int stats);
\end{verbatim}

currmode should have an integer in \([0, 1]\) to represent current execution mode as follows.

- 0 means we are in primary mode.
- 1 means we are in secondary mode.
stats inactivates statistics collection if 0, regardless the specification given by stats argument of oh1_init(). You may inactivate the statistics collection temporarily on, for example, the first call of oh1_transbound() for initial load balancing as discussed in Section 3.8.

return value is an integer in \([-1, 1]\) to represent the execution mode in the next simulation step as follows.

- -1 means helptand-helper configuration is (re)build and thus we will be in secondary mode. This also means that you have to broadcast field-arrays from helptands to their helpers if their replications are necessary for helpers.
- 0 means we will be in primary mode.
- 1 means we will be in secondary mode but helptand-helper configuration has been kept.

Usually, telling the current execution mode and receiving that in the next simulation step to/from oh1_transbound() is easily implemented by having your own currmode variable. That is, the following should be necessary and sufficient.

- Give 0 to oh1_transbound() at the first call because you have not yet built helptand-helper configuration even if the initial particle distribution causes an unacceptable load imbalance.

- Let your own currmode be the return value. If it is negative, let it be 1 and broadcast field-arrays if necessary. Then give it to oh1_transbound() on the second call and repeat this for successive calls.
3.4.3 oh1_broadcast()

The function (subroutine) oh1_broadcast() performs red-black broadcast communications in the families the local node belongs to. The arguments of the function pbuf, pcount and ptype specify the data to be broadcasted in the primary family which the local node belongs to as the helpand, while sbuf, scount and stype are for the data to be broadcasted in the secondary family which the local node belongs to as a helper, as shown in Figure 7. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary broadcast if it is a leaf and the secondary one if it is the root.

Fortran Interface

```fortran
subroutine oh1_broadcast(pbuf, sbuf, pcount, scount, ptype, stype)
  implicit none
  real*8, intent(in) :: pbuf
  real*8, intent(out) :: sbuf
  integer, intent(in) :: pcount
  integer, intent(in) :: scount
  integer, intent(in) :: ptype
  integer, intent(in) :: stype
end subroutine
```

C Interface

```c
void oh1_broadcast(void *pbuf, void *sbuf, int pcount, int scount,
                   MPI_Datatype ptype, MPI_Datatype stype);
```

pbuf should be (the pointer to) the first element of the data buffer which the local node broadcasts to its helpers in its primary family.

sbuf should be (the pointer to) the first element of the data buffer to receive data broadcasted in the secondary family.

pcount should have the number of ptype elements to be broadcasted in the primary family. This value should match scount of the call in the helpers.

scount should have the number of stype elements to be broadcasted in the secondary family. This value should match pcount of the call in the helpand.

ptype should have the MPI data-type of elements to be broadcasted in the primary family. This value should match stype of the call in the helpers.

stype should have the MPI data-type of elements to be broadcasted in the secondary family. This value should match ptype of the call in the helpand.

---

7 In the Fortran module file `oh_mod1.F90`, the arguments pbuf and sbuf of oh1_broadcast, oh1_all_reduce and oh1_reduce are declared as real*8 type hoping it matches the type of the elements in your buffers. If this is incorrect, feel free to modify the declaration or to remove it, so that your compiler accept your calls of the library subroutines.
3.4.4 oh1_all_reduce()

The function (subroutine) oh1_all_reduce() performs red-black all-reduce communications in the families the local node belongs to. The arguments of the function pbuf, pcount, ptype, pop specify the data to be reduced in the primary family, while sbuf, scount, stype and sop for the data to be reduced in the secondary family. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary reduction if it is a leaf and the secondary one if it is the root.

Fortran Interface

```fortran
subroutine oh1_all_reduce(pbuf, sbuf, pcount, scount, ptype, stype, &
   pop, sop)
   implicit none
   real*8,intent(inout) :: pbuf
   real*8,intent(inout) :: sbuf
   integer,intent(in) :: pcount
   integer,intent(in) :: scount
   integer,intent(in) :: ptype
   integer,intent(in) :: stype
   integer,intent(in) :: pop
   integer,intent(in) :: sop
end subroutine
```

C Interface

```c
void oh1_all_reduce(void *pbuf, void *sbuf, int pcount, int scount,
   MPI_Datatype ptype, MPI_Datatype stype,
   MPI_Op pop, MPI_Op sop);
```

pbuf should be (the pointer to) the first element of the data buffer to be reduced in the primary family. The buffer is replaced with the reduction result.

sbuf should be (the pointer to) the first element of the data buffer to be reduced in the secondary family. The buffer is replaced with the reduction result.

pcount should have the number of ptype elements to be reduced in the primary family. This value should match scount of the call in the helpers.

scount should have the number of stype elements to be reduced in the secondary family. This value should match pcount of the call in the helpers.

ptype should have the MPI data-type of elements to be reduced in the primary family. This value should match stype of the call in the helpers.

stype should have the MPI data-type of elements to be reduced in the secondary family. This value should match ptype of the call in the helpers.

pop should have the MPI operator for the reduction in the primary family. This value should match sop of the call in the helpers.

sop should have the MPI operator for the reduction in the secondary family. This value should match pop of the call in the helpers.
3.4.5 oh1_reduce()

The function (subroutine) oh1_reduce() performs red-black simple one-way reduce communications in the families the local node belongs to. The arguments of the function pbuf, pcount, ptype, pop specify the data to be reduced in the primary family, while sbuf, scount, stype and sop for the data to be reduced in the secondary family. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary reduction if it is a leaf and the secondary one if it is the root.

**Fortran Interface**

```fortran
subroutine oh1_reduce(pbuf, sbuf, pcount, scount, ptype, stype, pop, sop)
    implicit none
    real*8,intent(inout) :: pbuf
    real*8,intent(in) :: sbuf
    integer,intent(in) :: pcount
    integer,intent(in) :: scount
    integer,intent(in) :: ptype
    integer,intent(in) :: stype
    integer,intent(in) :: pop
    integer,intent(in) :: sop
end subroutine
```

**C Interface**

```c
void oh1_reduce(void *pbuf, void *sbuf, int pcount, int scount,
    MPI_Datatype ptype, MPI_Datatype stype,
    MPI_Op pop, MPI_Op sop);
```

- **pbuf** should be (the pointer to) the first element of the data buffer to be reduced in the primary family. The buffer is replaced with the reduction result.

- **sbuf** should be (the pointer to) the first element of the data buffer to be reduced in the secondary family. The buffer will remain unchanged.

- **pcount** should have the number of ptype elements to be reduced in the primary family. This value should match scount of the call in the helpers.

- **scount** should have the number of stype elements to be reduced in the secondary family. This value should match pcount of the call in the helpers.

- **ptype** should have the MPI data-type of elements to be reduced in the primary family. This value should match stype of the call in the helpers.

- **stype** should have the MPI data-type of elements to be reduced in the secondary family. This value should match ptype of the call in the helpers.

- **pop** should have the MPI operator for the reduction in the primary family. This value should match sop of the call in the helpers.

- **sop** should have the MPI operator for the reduction in the secondary family. This value should match pop of the call in the helpers.
3.5 Level-2 Library Functions

Level-2 library provides the following functions.

- **oh2_init()**: Performs initialization similar to what **oh1_init()** does and that of level-2’s own for particle buffers.

- **oh2_max_local_particles()**: Calculates the size of particle buffers.

- **oh2_transbound()**: Performs load balancing similar to **oh1_transbound()** and transfers particles according to the schedule.

- **oh2_inject_particle()**: Injects a particle to the bottom of the particle buffer.

The function API for Fortran programs is given by the module named **ohhelp2** in the file **oh_mod2.F90**, while API for C is embedded in **ohhelp.c.h**.

3.5.1 Particle Data Type

Since **oh2_transbound()** transfers particles among nodes, it needs to know how each particle is represented. The default configuration of the **struct** to represent a particle for C-coded simulator body and the library, namely **S_particle** is defined in the C header file **oh_part.h**, while its Fortran counterpart **oh_particle** is given in **oh_type.F90**. Both definitions are of course consistent with the following elements.

- **x**, **y** and **z** are for the *x/y/z* coordinates of the point at which a particle resides.
- **vx**, **vy** and **vz** are for the *x/y/z* components of the velocity of a particle.
- **pid** is the unique identifier of a particle by which, for example, you can trace the trajectory of the particle.
- **nid** is the identifier of the subdomain in which a particle resides.
- **spec** is the identifier of the species which a particle belongs to.

In the elements listed above, **nid** is essential for the library and **spec** is also necessary if \( S > 1 \) and you inject particles by the library function **oh2_inject_particle()**. Therefore, you may freely modify the definitions in **oh_part.h** and, if your simulator is coded in Fortran, **oh_type.F90**, by adding, removing and/or renaming other elements. As for **spec**, you can **union-ize** it with, for example, **nid** in **oh_part.h**, if \( S = 1 \) or you do not use **oh2_inject_particle()** and want to save four bytes for each particle.

The verbatim definitions of **S_particle** and **oh_particle** are as follows.

```c
struct S_particle {
    double x, y, z, vx, vy, vz;
    long long pid;
    int nid, spec;
};

type oh_particle
    sequence
        real*8 :: x, y, z, vx, vy, vz
        integer*8 :: pid
        integer :: nid, spec
end type
```

30
3.5.2 oh2_init()

The function (subroutine) oh2_init() receives a few fundamental parameters and arrays through which oh2_transbound() interacts with your simulator body. It also initializes internal data structures used in level-1 and level-2 library. Among its fourteen arguments, other library functions directly refer to only the bodies of the arguments nphgram and pbuf as their implicit inputs. Therefore, after the call of oh1_init(), modifying the bodies of other arguments has no effect to library functions.

**Fortran Interface**

```fortran
subroutine oh2_init(sdid, nspec, maxfrac, nphgram, totalp, &
                   pbuf, pbase, maxlocalp, mycomm, nbor, pcoord, &
                   stats, repiter, verbose)
  use oh_type
  implicit none
  integer,intent(out) :: sdid(2)
  integer,intent(in) :: nspec
  integer,intent(in) :: maxfrac
  integer,intent(inout) :: nphgram(:, :, :)
  integer,intent(out) :: totalp(:, :)
  type(oh_particle),intent(inout) :: pbuf(:)
  integer,intent(out) :: pbase(3)
  integer,intent(in) :: maxlocalp
  type(oh_mycomm),intent(out) :: mycomm
  integer,intent(inout) :: nbor(3, 3, 3) ! for 3D codes.
  integer,intent(in) :: pcoord(OH_DIMENSION)
  integer,intent(in) :: stats
  integer,intent(in) :: repiter
  integer,intent(in) :: verbose
end subroutine
```

**C Interface**

```c
void oh2_init(int **sdid, int nspec, int maxfrac, int **nphgram,
               int **totalp, struct S_particle **pbuf, int **pbase,
               int maxlocalp, void *mycomm, int **nbor,
               int *pcoord, int stats, int repiter, int verbose);
```

sdid
nspec
maxfrac
nphgram
totalp

See Section 3.4.1 because the arguments above are perfectly equivalent to those of oh1_init().

pbuf($P_{lim}$) (for Fortran)
**pbuf** (for C)

The argument pbuf should be an one-dimensional array of oh_particle type structure elements in Fortran, while it should be a double pointer to an array of S_particle structure in C. The array have to be large enough to accommodate $P_{lim}$ particles, where $P_{lim}$ is given through the argument maxlocalp and should not be less than 31.
Figure 8: Particle buffer and related variables.

$P_{\text{max}}$ at any time (Figure 8). In C code, \texttt{pbuf} can be a pointer to NULL (not NULL itself) to make \texttt{oh2\_init()} allocate the buffer for you and return the pointer to it through the argument.

\texttt{pbase(3)} (for Fortran)

\texttt{**pbase} (for C)

The argument \texttt{pbase} should be an one dimensional array of three elements in Fortran, while it should be a double pointer to such an array in C. After zero-cleared by \texttt{oh2\_init()}, each call of \texttt{oh2\_transbound()} make the array for the local node $n$ have $0$, $Q_n^p$ and $Q_n$ in this order to represent the zero-origin displacement of the first primary particle and the first secondary particle, and the head of unused region of \texttt{pbuf}. That is, the first $Q_n^p$ portion of \texttt{pbuf} is used for primary particles, while the second $Q_n^{\text{parent}(n)} = Q_n - Q_n^p$ particles are for secondary particles. In C code, \texttt{pbuf} can be a pointer to NULL (not NULL itself) to make \texttt{oh2\_init()} allocate the array for you and return the pointer to it through the argument.

\texttt{maxlocalp} should have the absolute limit of the particle buffer \texttt{pbuf} and thus defines $P_{\text{lim}}$. You may ask the library function \texttt{oh2\_max\_local\_particles()} to calculate $P_{\text{lim}}$ from the system-wide absolute limit. Note that \texttt{oh2\_init()} allocates a buffer for particle transfer and thus your machine should have memory large enough to have $2 \times P_{\text{lim}}$ particles per computation node.

\texttt{mycomm}

\texttt{nbcor}

\texttt{pcord}

\texttt{stats}

\texttt{reptier}

\texttt{verbose}

See Section 3.4.1 because the arguments above are perfectly equivalent to those of \texttt{oh1\_init()}.

Note that \texttt{oh2\_init()} has neither arguments \texttt{rcounts} nor \texttt{scounts} which \texttt{oh1\_init()} has, because particle transfer in \texttt{oh2\_transbound()} makes it unnecessary to report transfer schedule.

3.5.3 \texttt{oh2\_max\_local\_particles()}

The function \texttt{oh2\_max\_local\_particles()} calculates the absolute maximum number of particles which a node can accommodate and returns it to its caller. The return value can be directly passed to the argument \texttt{maxlocalp} of \texttt{oh2\_init()}. 
**Fortran Interface**

```fortran
integer function oh2_max_local_particles(npmax, maxfrac, minmargin)
imPLICIT none
integer*8 , intent(in) :: npmax
integer , intent(in) :: maxfrac
integer , intent(in) :: minmargin
end function
```

**C Interface**

```c
int oh2_max_local_particles(long long int npmax, int maxfrac, int minmargin);
```

npmax should be the absolute maximum number of particles which your simulator is capable of as a whole.

maxfrac should have the tolerance factor percentage of load imbalance $\alpha$ and should be same as the argument maxfrac of oh2_init().

minmargin should be the minimum margin by which the return value $P_{lim}$ has to clear over the per node average of npmax.

**Return Value** is the number of particles $P_{lim}$ given by the following.

$$P_{lim} = \left\lfloor \frac{npmax}{N} \right\rfloor \quad P_{lim} = \max(\left\lfloor P(N + \alpha)/100 \right\rfloor, P + \text{minmargin})$$

Note that minmargin is the margin over $P$ to be kept besides the tolerance factor $\alpha$ for, e.g., initial particle accommodation in each node. Therefore it does not assure that a node has a room for minmargin particles in simulation. If you need such a room for, e.g., particle injection, add the room to $P_{lim}$ to give it the argument maxlocalp of oh2_init().

### 3.5.4 oh2_transbound()

The function oh2_transbound() at first performs operations for load balancing as same as that oh1_transbound() does; examination of nphgram to check the balancing and (re)building of helpand-helper configuration updating mycomm if necessary. Then, instead of reporting the particle transfer schedule, it sends particles in pbuf to other nodes and receives them into pbuf, updates totalp and pbase according to the transfer result, and clears nphgram with zeros. Note that the arrays nphgram, pbuf, totalp and pbase and the structure mycomm were given to oh2_init() as its arguments.

The arguments of oh2_transbound() and its return value, besides these global arrays and structures, are perfectly equivalent to those of oh1_transbound() and thus see Section 3.4.2 for them.

**Fortran Interface**

```fortran
integer function oh2_transbound(currmode, stats)
imPLICIT none
integer , intent(in) :: currmode
integer , intent(in) :: stats
end function
```
C Interface

int oh2_transbound(int currmode, int stats);

3.5.5 oh2_inject_particle()

The function (subroutine) oh2_inject_particle() injects a given particle at the bottom of pbuf and increase an element of nphgram according to its residence subdomain and species. Note that the number of particles injected in a simulation step should not be greater than \( P_{\text{lim}} - Q_n \).

Fortran Interface

```fortran
subroutine oh2_inject_particle(part)
  use oh_type
  implicit none
  type(oh_particle),intent(in) :: part
end subroutine
```

C Interface

```c
void oh2_inject_particle(struct S_particle *part);
```

part (for Fortran)
*part (for C)

The argument part should be a oh_particle structure in Fortran, or a pointer to S_particle structure in C, to be injected. Elements in the given particle structure should be completely set with significant values in advance, especially for nid and spec elements which are referred to by the function to update nphgram. See Section 3.7 for further discussion on injection.

3.6 Level-3 Library Functions

Level-3 library provides the following functions.

oh3_init() performs initialization similar to what oh2_init() does and that of level-3 own for communications of field-arrays.

oh13_init() performs initialization similar to what oh3_init() does but excludes that for the particle buffer. That is, roughly speaking, oh13_init() is equal to oh3_init() minus oh2_init() plus oh1_init().

oh3_transbound() performs load balancing almost equivalent to oh2_transbound() or oh1_transbound() depending on the initializer you choose.

oh3_map_particle_to_neighbor() finds the subdomain which will be the resident of a boundary crossing particle and is neighboring to the primary or secondary subdomain of the local node, and then returns its identifier.

oh3_map_particle_to_subdomain() finds the subdomain which will be the resident of a boundary crossing particle and may be anywhere in the whole space domain, and then returns its identifier.
oh3_bcast_field() performs broadcast communication of a field-array in helpand-helper families.

oh3_allreduce_field() performs all-reduce communication of a field-array in helpand-helper families.

oh3_reduce_field() performs simple one-way reduce communication of a field-array in helpand-helper families.

oh3_exchange_borders() performs neighboring communication to exchange subdomain boundary data of a field-array.

The function API for Fortran programs is given by the module named ohhelp3 in the file oh_mod3.F90, while API for C is embedded in ohhelp_c.h.

3.6.1 oh3_init()

The function (subroutine) oh3_init() receives a number of fundamental parameters and arrays through which oh3_transbound() interacts with your simulator body. It also initializes internal data structures used in level-1, level-2 and level-3 library. Among its twenty three (23!!!) arguments, other library functions directly refer to only the bodies of the arguments nphgram and pbuf as their implicit inputs. Therefore, after the call of oh1_init(), modifying the bodies of other arguments has no effect to library functions.

**Fortran Interface**

```fortran
subroutine oh3_init(sdid, nspec, maxfrac, nphgram, totalp, &
  pbuf, pbase, maxlocalp, mycomm, nbor, pcoord, &
  sdoms, scoord, nbound, bcond, bounds, ftypes, &
  cfields, ctypes, fsizes, &
  stats, repiter, verbose)
use oh_type
implicit none
integer,intent(out) :: sdid(2)
integer,intent(in) :: nspec
integer,intent(in) :: maxfrac
integer,intent(inout) :: nphgram(:,,:,:)
integer,intent(out) :: totalp(:,:)
type(oh_particle),intent(inout) :: pbuf(:)
integer,intent(out) :: pbase(3)
integer,intent(in) :: maxlocalp
integer,intent(in) :: mycomm
integer,intent(inout) :: nbor(3,3,3) ! for 3D codes.
integer,intent(in) :: pcoord(OH_DIMENSION)
integer,intent(inout) :: sdoms(:,,:,:)
integer,intent(in) :: scoord(2,OH_DIMENSION)
integer,intent(in) :: nbound
integer,intent(in) :: bcond(2,OH_DIMENSION)
integer,intent(inout) :: bounds(:,,:,:)
integer,intent(in) :: ftypes(:,:)
integer,intent(in) :: cfields(:)
integer,intent(in) :: ctypes(:,,:,:)
integer,intent(out) :: fsizes(:,,:,:)
integer,intent(in) :: stats
```

35
integer, intent(in) :: repiter
integer, intent(in) :: verbose
end subroutine

sdid
nspec
maxfrac
nphgram
totalp

See Section 3.4.1 because the arguments above are perfectly equivalent to those of oh1_init().

pbuf
pbase
maxlocalp

See Section 3.5.2 because the arguments above are perfectly equivalent to that of oh2_init().

mycomm
nbor
pcoord

See Section 3.4.1 because the arguments above are perfectly equivalent to those of oh1_init().

sdoms(2, D, N) should be an array whose element sdoms(β, d, m+1) should have the d-th (d ∈ [1, D]) dimensional integer coordinate of the lower (β = 1) or upper (β = 2) boundary of the subdomain m ∈ [0, N−1], namely δ_x(m), δ_y(m), δ_z(m) and (δ^u_x(m)−1, δ^u_y(m)−1, δ^u_z(m)−1), the subarray sdoms(1:2, 1:3, m+1) should have the followings (Figure 9).

sdoms(1:2, 1:3, m+1) = reshape((/δ^l_x(m), δ^u_x(m), δ^l_y(m), δ^u_y(m), δ^l_z(m), δ^u_z(m)/), (/2,3/))

Note that if the subdomain m is the d-th dimensional lower (upper) neighbor of n sharing a (D−1)-dimensional plane perpendicular to d-th axis (e.g., a neighbor along x-axis sharing a yz-plane), n’s lower (upper) boundary plane has to be m’s upper (lower) boundary plane. For example, if D = 3 and m is n’s lower neighbor along x-axis, the following must be satisfied.

\[
\Delta^l_x = \min_{m \in [0, N-1]} \{\delta^l_x(m)\} \quad \Delta^u_x = \max_{m \in [0, N-1]} \{\delta^u_x(m)\} \\
\delta^l_x(n) = \delta^u_x(m) \lor \delta^l_x(n) = \delta^u_x(m) - (\Delta^u_x - \Delta^l_x) \lor \delta^l_x(n) = \delta^u_x(m) + (\Delta^u_x - \Delta^l_x) \land \\
\delta^l_y(n) = \delta^l_y(m) \land \delta^u_y(n) = \delta^u_y(m) \land \delta^l_z(n) = \delta^l_z(m) \land \delta^u_z(n) = \delta^u_z(m)
\]

Alternatively, if the work to define sdoms is bothersome for you, you may delegate it to oh3_init() by making sdoms(1,1,1) > sdoms(2,1,1), and giving the lower and upper boundaries of the whole space domain [Δ^l_1, Δ^u_1−1] × ... × [Δ^l_D, Δ^u_D−1] through the argument array scoord(2, D) as follows.

scoord(:, :) = reshape((/sdoms(1,2,1:3,1:3,1+1) := reshape((/Δ^l_1, Δ^u_1, ..., Δ^l_D, Δ^u_D/), (2, D))

36
Figure 9: \texttt{sdoms} and its default setting for space domain of 27 × 19 given by \texttt{scord} and node coordinate system of 5 × 3 given by \texttt{pcoord}.

In this case, \texttt{oh3\_init()} also refers to the argument array \texttt{pcoord(D)=/II_1, \ldots, II_D/} and defines \texttt{sdoms(\beta, d, m + 1)} for \( m = \text{rank}(\pi_1, \ldots, \pi_D) \) as follows.

\[
\begin{align*}
a_d &= \lfloor (\Delta^u_d - \Delta^l_d) / \Pi_d \rfloor \\
k_d &= \Pi_d - ((\Delta^u_d - \Delta^l_d) \mod \Pi_d)
\end{align*}
\]

\[
\text{sdoms}(1, d, m + 1) = \begin{cases} \\
\Delta^u_d + \pi_d \cdot a_d & \pi_d \leq k_d \\
\Delta^l_d + \pi_d \cdot a_d + (\pi_d - k_d) & \pi_d > k_d
\end{cases} \quad \text{for } \pi_d \leq k_d
\]

\[
m_d^+ = \text{rank}(\pi_1, \ldots, \pi_d + 1, \ldots, \pi_D)
\]

\[
\text{sdoms}(2, d, m + 1) = \begin{cases} \\
\text{sdoms}(1, d, m_d^+ + 1) & \pi_d < \Pi_d - 1 \\
\Delta^u_d & \pi_d = \Pi_d - 1
\end{cases} \quad \text{for } \pi_d > k_d
\]

That is, if we have \( \Pi_x \) subdomains along \( x \)-axis and the lower and upper boundaries of the whole domain along \( x \)-axis are \( \Delta^l_x \) and \( \Delta^u_x \), easternmost \((\Delta^u_x - \Delta^l_x) \mod \Pi_x\) subdomains have \( x \)-edges of \( [(\Delta^u_x - \Delta^l_x) / \Pi_x] \) while remaining western ones have \( x \)-edges of \( [(\Delta^u_x - \Delta^l_x) / \Pi_x] \). Note that the delegation of setting \texttt{sdoms(:, :,:)} also means that for the argument array \texttt{bounds(:,,:,:)}.
Figure 10: Complicated subdomains and their boundaries with walls and holes.

`scoord(2,D)` should be an array whose element `scoord(β,d)` has the $d$-th ($d \in [1,D]$) dimensional integer coordinate of the lower ($β = 1$) or upper ($β = 2$) boundary of the whole space domain, if you delegate the setting of the array `sdoms(2,D,N)` to `oh3_init()`. That is, `scoord(1:2,1:D)` should have the following for the space domain of $[Δ^1_1, Δ^u_1−1] \times \ldots \times [Δ^1_D, Δ^u_D−1]$.

```
scoord(:, :) = reshape((/$Δ^1_1, Δ^u_1, \ldots, Δ^1_D, Δ^u_D/$), (2, D))
```

Otherwise, i.e., if you completely specify `sdoms(:, :, :)` by yourself, the array can have any values.

`nbound` should be a positive integer representing the number of boundary condition types $B$ of the space domain. That is, you can specify a type of boundary condition $b \in [1,B]$ for each boundary of the whole space domain through the argument `bcond(2,D)` or of each subdomain through the argument `bounds(2,D,N)`. Then also you can specify how the communication through a boundary of a specific type is performed through the argument `ctypes(3,2,B,C)`. Remember that the boundary condition type 1 is special and reserved for periodic boundaries.

`bcond(2,D)` should be an array whose element `bcond(β,d)` has the type of boundary condition $b \in [1,B]$ for the lower ($β = 1$) or upper ($β = 2$) boundary plane of the whole space domain perpendicular to the $d$-th axis, if you delegate the setting of the array `sdoms(2,D,N)` and `bounds(2,D,N)` to `oh3_init()`. Otherwise, the array can have any values.

`bounds(2,D,N)` should be an array whose element `bounds(β,D,m+1)` has the type of boundary condition $b \in [1,B]$ for the lower ($β = 1$) or upper ($β = 2$) boundary plane of the subdomain $m$ perpendicular to the $d$-th axis, if you specify `sdoms(:, :, :)` by yourself. Remember that, for a pair of adjacent subdomains, the boundary condition of the boundary plane shared by them must have type 1, unless the plane is a special wall. Also remember that a subdomain boundary, which is also a boundary of the whole space domain with periodic boundary condition, should have type 1 too. See Figure 10 an example of complicated subdomain boundaries with walls and holes.

Otherwise, i.e., you delegate the setting of the array `sdoms(2,D,N)` to `oh3_init()`, it is assumed that you also delegate the setting of `bounds(:, :, :)`. In this case, `oh3_init()` gives the type 1 to internal boundaries, while external boundaries of the whole space domain will have corresponding types specified by `bcond(:, :)` as shown in Figure 11.
ftypes(7,F+1) should be an array whose elements ftypes(1:7,f) should have the followings to specify the field-array associated to grid points in a subdomain and identified by the integer \( f \in [1,F] \), while ftypes(1,F+1) should be 0 (or less) to tell oh3_init() that you have \( F \) types of arrays.

ftypes(1,f) is the number of elements associated to a grid point of a type \( f \) field-array. For example, if \( f \) is for electromagnetic field array namely \( \text{eb}(6,:,;,:,2) \) whose first dimension is for three electric and three magnetic field vector components, ftypes(1,f) should be 6.

ftypes(2:3,f) defines lower (2) and upper (3) extensions \( \sigma_l(f) \) and \( \sigma_u(f) \) required for the type \( f \) field-arrays, besides extensions for communication. That is, for a subdomain of \([0,\sigma_1-1] \times \cdots \times [0,\sigma_D-1] \), the array for \( f \) is at least as large as:

\[
(\sigma_l(f):\sigma_1+\sigma_u(f)-1,\ldots,\sigma_l(f):\sigma_D+\sigma_u(f)-1)
\]

Note that if the field-arrays of type \( f \) do not need such non-communicational extensions, you should let \( \sigma_l(f) = \sigma_u(f) = 0 \).

ftypes(4:5,f) defines lower (4) and upper (5) extensions \( e_l(f) \) and \( e_u(f) \) for the broadcast of the type \( f \) field-arrays. For example, for your electromagnetic filed \( \text{eb}(6,:,;,:,2) \) of type \( f \) for a subdomain of \([0,\sigma_x-1] \times [0,\sigma_y-1] \times [0,\sigma_z-1] \), oh3_bcast_field() sends elements in the range\(^8\):

from \( \text{eb}(1,e_l(f),e_l(f),\sigma_l(f),1) \)
to \( \text{eb}(6,\sigma_x+e_u(f)-1,\sigma_y+e_u(f)-1,\sigma_z+e_u(f)-1,1) \)

to the helpers of the local node(Figure 12). Note that if the field-arrays of type \( f \) are never broadcasted, you should let \( e_l(f) = e_u(f) = 0 \).

ftypes(6:7,f) defines lower (6) and upper (7) extensions \( c_l(f) \) and \( c_u(f) \) for the reduction of the type \( f \) field-array. For example, for your current density array of type \( f \) namely \( \sigma \), for the type \( f \) field-arrays, besides extensions for communication. That is, for a subdomain of \([0,\sigma_1-1] \times \cdots \times [0,\sigma_1-1] \), oh3_allreduce_field() or oh3_reduce_field() performs the reduction of the elements in the range\(^9\):

from \( \text{cd}(1,c_l(f),c_l(f),\sigma_l(f),1) \)
to \( \text{cd}(3,\sigma_x+c_u(f)-1,\sigma_y+c_u(f)-1,\sigma_z+c_u(f)-1,1) \)

\(^8\)Not the subarray \( \text{eb}(:,c_l(f):\sigma_z+e_u(f)-1,c_u(f):\sigma_y+e_u(f)-1,c_u(f):\sigma_u+e_u(f)-1,1) \).

\(^9\)Not the subarray \( \text{cd}(:,c_l(f):\sigma_z+e_u(f)-1,c_u(f):\sigma_y+e_u(f)-1,c_u(f):\sigma_u+e_u(f)-1,1) \).

---

**Figure 11:** Default setting of subdomain boundaries.
Figure 12: Type \( f \) field-array of \((x^s + 5) \times (y^s + 5)\) for a subdomain of \([0, x^s - 1] \times [0, y^s - 1]\) and its elements (painted) broadcasted by \texttt{oh3\_bcast\_field()} with setting of \( e^y_\text{b}(f) = -1 \) and \( e^x_\text{b}(f) = 2 \).

to have the sum in the primary family of the local node. Note that if you will never perform reductions on the field-arrays of type \( f \), you should let \( e^y_\text{r}(f) = e^x_\text{r}(f) = 0 \).

c\texttt{fields}(C+1) should be an array whose element \texttt{cfields}(c) has \( f \in [1, F] \) to identify a field-array type for which a type of boundary communication identified by the integer \( c \in [1, C] \) is defined, while \texttt{c\texttt{types}(C+1)} should be 0 (or less) to tell \texttt{oh3\_\texttt{init}()} that you have \( C \) types of boundary communications.

This array implies that a field-array may have two or more boundary communication types according to the timing of the communication, or no boundary communication may be taken for the field-array.

c\texttt{types}(3,2,B,C) should be an array whose element \texttt{c\texttt{types}(1:3,w,b,c)=(/e_f,e_t,s/)\texttt{defines downward (}w=1\texttt{)} or upward (}w=2\texttt{)} boundary communication through the boundary of type \( b \in [1, B] \) for a field-array \( f = \texttt{c\texttt{fields}(c)} \) of the subdomain of \([0, \sigma_1 - 1] \times \ldots [0, \sigma_D - 1]\) as follows (Figure 13).

- Downward (\( w = 1 \)) communication along \( d \)-th dimensional axis is the pair of sending \( s \) planes perpendicular to the axis to the lower neighbor and receiving the planes from the upper neighbor. The first plane to be sent has \( d \)-th dimensional coordinate \( e_f \), while that to be received is at \( \sigma_d + e_t \).
- Upward (\( w = 2 \)) communication along \( d \)-th dimensional axis is the pair of sending \( s \) planes perpendicular to the axis to the upper neighbor and receiving the planes from the lower neighbor. The first plane to be sent has \( d \)-th dimensional coordinate \( \sigma_d + e_f \), while that to be received is at \( e_t \).

Therefore, when you just need \( s_l \) and \( s_u \) planes at the lower and upper boundaries surrounding a subdomain, \( e_f = e_t = 0 \) and \( s = s_u \) for downward communication, while \( e_f = e_t = -s_l \) and \( s = s_l \) for upward communication as shown in Figure 13(b). On the other hand, if you need these planes keeping those calculated by the local node for, e.g., the addition of current densities at boundaries, \( e_f = e_t + s_u \) and \( s = s_u \) for downward communication, while \( e_f = e_t + s_l \) for upward communication, as shown in Figure 14.
Figure 13: Field-array with downward communication \((e_f, e_t, s) = (0, 0, 2)\) and upward communication \((e_f, e_t, s) = (-1, -1, 1)\).

Figure 14: Field-array with downward communication \((e_f, e_t, s) = (-1, 2, 3)\) and upward communication \((e_f, e_t, s) = (-1, -4, 3)\).
Note that if no data is transferred by downward and/or upward type c communication through a boundary of type b, the element ctypes(3, w, b, c), i.e., s, should be set to 0.

fsizes(2, D, F) should be an array whose element fsizes(β, d, f) will have φ_d(β) (β = 1) or φ_d(β) − 1 (β = 2) for the field-arrays of type f to notify you that the field-arrays must have the shape (ε, φ_1(β−1),...,φ_D(β−1)) for the leading D+1 dimensions, where ε = ftypes(1, f). That is, if D = 3 and your field-array for electromagnetic field vectors eb(6, :, :, :, 2) has type feb, you have to allocate the array by the following.

allocate(eb(6, fsizes(1, 1, feb)+1:fsizes(2, 1, feb)),
fsizes(1, 2, feb):fsizes(2, 2, feb),
fsizes(1, 3, feb):fsizes(2, 3, feb), 2))

Note that the allocation above makes the origin of subdomains eb(:, 0, 0, 0, :). Therefore, if you like to define some other coordinates to the origin, for example eb(:, 1, 2, 3, :), you have to do the following keeping the number of elements in each dimension.

allocate(eb(6, fsizes(1, 1, feb)+1:fsizes(2, 1, feb)+1,
fsizes(1, 2, feb)+2:fsizes(2, 2, feb)+2,
fsizes(1, 3, feb)+3:fsizes(2, 3, feb)+3, 2))

The value of φ_d(f) and φ_d(f) are calculated by the followings to obtain the maximum extensions at lower and upper boundaries from ftypes(:, :, :), cfields(): and ctypes(:, :, :, :), and the maximum size of each subdomain edge from sdoms(:, :, :).

\[ \Gamma(f) = \{ c | c\text{fields}(c) = f \} \]
\[ \lambda(c, s) = \begin{cases} c & s \neq 0 \\ 0 & s = 0 \end{cases} \]
\[ s^l(b, c) = \text{ctypes}(3, 1, b, c) \]
\[ s^l(b, c) = \text{ctypes}(3, 2, b, c) \]
\[ e^l_T(b, c) = \lambda(\text{ctypes}(1, 1, b, c), s^l(b, c)) \]
\[ e^l_T(b, c) = \lambda(\text{ctypes}(2, 1, b, c), s^l(b, c)) \]
\[ e^l_T(b, c) = \lambda(\text{ctypes}(1, 2, b, c), s^l(b, c)) \]
\[ e^l_T(b, c) = \lambda(\text{ctypes}(2, 2, b, c), s^l(b, c)) \]
\[ e^l_T(f) = \min_{b \in [1, B], c \in \Gamma(f)} \{ e^l_T(b, c) + s^l(b, c) \} \]
\[ e^l_T(f) = \max_{b \in [1, B], c \in \Gamma(f)} \{ e^l_T(b, c) + s^l(b, c) \} \]
\[ \delta_{d}^{\text{max}}(f) = \max_{m \in [0, N-1]} \{ \delta_{d}^{\text{max}}(m) \} \]
\[ \phi_{d}^{\text{max}}(f) = \phi_{d}^{\text{max}} + e_{d}^{\text{max}}(f) \]

For example, suppose D = 2, the subdomain decomposition is done as shown in Figure 9 with fully periodic boundaries, and you specify the followings for your elec-
tromagnetic field array $eb(6,:,1,2)$ with field-array type identifier $feb$ and boundary communication type identifier $ceb$.

\[
\text{ftypes}(::,feb)=\langle 6, 0, 0, 0, 1, 0, 0/ \rangle \\
\text{cfields}(ceb)=feb \\
\text{ctypes}(::,1,ceb)=\text{reshape}(\langle /0,0,2,-1,-1,1/ \rangle, (/3,2)/)
\]

Then you will have the followings in $fsizes(:,:,feb)$ to allocate the array by $allocate(eb(6,-1:8,-1:9,2))$.

\[
\begin{align*}
fsizes(1,1,feb) &= \min(\min(0,-1),0,0,0) = -1 \\
fsizes(2,1,feb) &= 6 + \max(\max(2,0),0,1,0) = 6 + 2 = 8 \\
fsizes(1,2,feb) &= \min(\min(0,-1),0,0,0) = -1 \\
fsizes(2,2,feb) &= 7 + \max(\max(2,0),0,1,0) = 7 + 2 = 9
\end{align*}
\]

\begin{verbatim}
stats repiter verbose
See Section 3.4.1 because the arguments above are perfectly equivalent to those of oh1_init().
\end{verbatim}

\begin{verbatim}
C Interface

void oh3_init(int **sdid, int nspec, int maxfrac, int **nphgram, 
int **totalp, struct S_particle **pbuf, int **pbase, 
int maxlocalp, void *mycomm, int **nbor, int *pcoord, 
int **sdoms, int *scoord, int nbound, int *bcond, int **bounds, 
int *ftypes, int *cfields, int *ctypes, int **fsizes, 
int stats, int repiter, int verbose);
\end{verbatim}

\begin{verbatim}
sdid nspec maxfrac nphgram
See Section 3.4.1 because the arguments above are perfectly equivalent to those of oh1_init().
\end{verbatim}

\begin{verbatim}
totalp pbuf pbase maxlocalp
See Section 3.5.2 because the arguments above are perfectly equivalent to that of oh2_init().
\end{verbatim}

\begin{verbatim}
mycomm nbor pcoord
See Section 3.4.1 because the arguments above are perfectly equivalent to those of oh1_init().
\end{verbatim}
**sdoms** should be a double pointer to an array of $N \times D \times 2$ elements to form \texttt{sdoms}[N][D][2] conceptually, or a pointer to \texttt{NULL} (not \texttt{NULL} itself) if you want the library to allocate and initialize the array and return the pointer to it through the argument. If you prepare the array, its element \texttt{sdoms}[m][d][β] should have the $d$-th ($d \in [0, D-1]$) dimensional integer coordinate of the lower ($β = 0$) or upper ($β = 1$) boundary of the subdomain $m \in [0, N-1]$, namely $δ^l_d(m)$ or $δ^u_d(m)$ respectively. For example, for the 3-dimensional cuboid subdomain $m$ whose grid points at west-south-east and east-north-top corners are $(δ^l_x(m), δ^l_y(m), δ^l_z(m))$ and $(δ^u_x(m)-1, δ^u_y(m)-1, δ^u_z(m)-1)$, the subarray \texttt{sdoms[m][][]} should have the followings (Figure 9).

\begin{align*}
\text{sdoms}[m][0][0] &= δ^l_x(m); & \text{sdoms}[m][0][1] &= δ^u_x(m); \\
\text{sdoms}[m][1][0] &= δ^l_y(m); & \text{sdoms}[m][1][1] &= δ^u_y(m); \\
\text{sdoms}[m][2][0] &= δ^l_z(m); & \text{sdoms}[m][2][1] &= δ^u_z(m);
\end{align*}

Note that if the subdomain $m$ is the $d$-th dimensional lower (upper) neighbor of $n$ sharing a $(D-1)$-dimensional plane perpendicular to $d$-th axis (e.g., a neighbor along $x$-axis sharing a $yz$-plane), $n$’s lower (upper) boundary plane has to be $m$’s upper (lower) boundary plane. For example, if $D = 3$ and $m$ is $n$’s lower neighbor along $x$-axis, the following must be satisfied.

\begin{align*}
Δ^l_x = \min_{m \in [0, N-1]} \{δ^l_x(m)\} \quad Δ^u_x = \max_{m \in [0, N-1]} \{δ^u_x(m)\} \\
(δ^l_y(n) = δ^l_y(m) \vee δ^l_z(n) = δ^l_z(m) - (Δ^u_x - Δ^l_x) \vee δ^l_z(n) = δ^u_y(m) + (Δ^u_x - Δ^l_x)) \wedge \\
δ^u_y(n) = δ^u_y(m) \wedge δ^u_z(n) = δ^u_z(m) \wedge δ^l_z(n) = δ^l_z(m)
\end{align*}

Alternatively, if the work to define \texttt{sdoms} is bothersome for you, you may delegate it to \texttt{oh3_init()} by passing a pointer to \texttt{NULL} or by making \texttt{sdoms[0][0][0]} > \texttt{sdoms[0][0][1]} and gives the lower and upper boundaries of the whole space domain $[Δ^l_0, Δ^u_0-1] \times \ldots \times [Δ^l_{D-1}, Δ^u_{D-1}]$ through the argument array \texttt{scord[D][2]} as follows.

\begin{verbatim}
int scord[D][2] = {{Δ^l_0, Δ^u_0}, \ldots, {Δ^l_{D-1}, Δ^u_{D-1}}};
\end{verbatim}

In this case, \texttt{oh3_init()} also refers to the argument array \texttt{pcoord[D] = \{Π_0, \ldots, Π_{D-1}\}} and defines \texttt{sdoms[m][d][β]} for $m = \text{rank}(π_0, \ldots, π_{D-1})$ as follows.

\begin{align*}
\text{sdoms}[m][d][0] &= \begin{cases} \\
\text{Δ}_d^l + π_d \cdot a_d & \text{if } π_d \leq k_d \\
\text{Δ}_d^l + π_d \cdot a_d + (\text{π}_d - k_d) & \text{if } π_d > k_d
\end{cases} \\
\text{sdoms}[m][d][1] &= \begin{cases} \\
\text{rank}(\text{π}_0, \ldots, \text{π}_d + 1, \ldots, \text{π}_{D-1}) & \text{if } π_d < Π_d - 1 \\
\text{Δ}_d^u & \text{if } Π_d - 1
\end{cases}
\end{align*}

That is, if we have $Π_d$ subdomains along $x$-axis and the lower and upper boundaries of the whole domain along $x$-axis are $Δ^l_d$ and $Δ^u_d$, eastmost $(Δ^u_d - Δ^l_d) \bmod Π_d$ subdomains have $x$-edges of $[(Δ^u_d - Δ^l_d)/Π_d]$ while remaining western ones have $x$-edges of $[(Δ^u_d - Δ^l_d)/Π_d]$. Note that the delegation of setting \texttt{sdoms} also means that for the argument array \texttt{bounds}. 44
*scoord should be a pointer to an array of \( D \times 2 \) to form \( \text{scoord}[D][2] \) conceptually, if you delegate the setting of the array \( \text{sdoms}[N][D][2] \) to **oh3_init**(). If so, its element \( \text{scoord}[d][\beta] \) should have the \( d \)-th \( (d \in [0, D-1]) \) dimensional integer coordinate of the lower \( (\beta = 0) \) or upper \( (\beta = 1) \) boundary of the whole space domain. That is, \( \text{scoord}[D][2] \) should have the following for the space domain of \( [\Delta u_1, \Delta u_{D-1}] \times \ldots \times [\Delta d_{B-1}, \Delta d_{B-1}] \).

\[
\text{int scoord}[D][2] = \{ \{\Delta l_1, \Delta l_{B} \}, \ldots, \{\Delta d_{B-1}, \Delta d_{B-1} \} \}
\]

Otherwise, i.e., if you completely specify \( \text{sdoms} \) by yourself, \( \text{scoord} \) can be NULL or the array can have any values.

*nbound* should be a positive integer representing the number of boundary condition types \( B \) of the space domain. That is, you can specify a type of boundary condition \( b \in [0, B-1] \) for each boundary of the whole space domain through the argument \( \text{bcond}[D][2] \) or of each subdomain through the argument \( \text{bounds}[N][D][2] \). Then also you can specify how the communication through a boundary of a specific type is performed through the argument \( \text{ctypes}[C][B][2][3] \). Remember that the boundary condition type 0 is special and reserved for periodic boundaries.

*bcond* should be a pointer to an array of \( D \times 2 \) to form \( \text{bcond}[D][2] \) conceptually, if you delegate the setting of the array \( \text{sdoms}[N][D][2] \) and \( \text{bounds}[N][D][2] \) to **oh3_init**(). If so, its element \( \text{bcond}[d][\beta] \) should have the type of boundary condition \( b \in [0, B-1] \) for the lower \( (\beta = 0) \) or upper \( (\beta = 1) \) boundary plane of the whole space domain perpendicular to the \( d \)-th axis. Otherwise, \( \text{bcond} \) can be NULL or the array can have any values.

**bounds** should be a double pointer to an array of \( N \times D \times 2 \) to form \( \text{bounds}[N][D][2] \) conceptually, if you specify \( \text{sdoms} \) by yourself. If so, its element \( \text{bounds}[m][d][\beta] \) should have the type of boundary condition \( b \in [0, B-1] \) for the lower \( (\beta = 0) \) or upper \( (\beta = 1) \) boundary plane of the subdomain \( m \) perpendicular to the \( d \)-th axis. Remember that, for a pair of adjacent subdomains, the boundary condition of the boundary plane shared by them must have type 0, unless the plane is a special wall. Also remember that a subdomain boundary, which is also a boundary of the whole space domain with periodic boundary condition, should have type 0 too. See Figure 10 an example of complicated subdomain boundaries with walls and holes.

Otherwise, i.e., you delegate the setting of the array \( \text{sdoms}[N][D][2] \) to **oh3_init**(), it is assumed that you also delegate the setting of \( \text{bounds} \). In this case, **oh3_init**() allocate the array of \( N \times D \times 2 \) and set the pointer to it to **bounds** if it was NULL, and then initialize \( \text{bounds} \) so that internal boundaries have the type 0, while external boundaries of the whole space domain have corresponding types specified by \( \text{bcond} \) as shown in Figure 11.

*ftypes* should be a pointer to an array of \( (F+1) \times 7 \) to form \( \text{ftypes}[F+1][7] \) conceptually. Its element \( \text{ftypes}[f][] \) should have the followings to specify the field-array associated to grid points in a subdomain and identified by the integer \( f \in [0, F-1] \), while \( \text{ftypes}[F][0] \) should be 0 (or less) to tell **oh3_init**() that you have \( F \) types of arrays.

\( \text{ftypes}[f][0] \) is the number of elements associated to a grid point of a type \( f \) field-array. For example, if \( f \) is for electromagnetic field array namely eb[2][1][1][1][1][1][1]
of six double elements struct for three electric and three magnetic field vector components, ftypes[f][0] should be 6.

ftypes[f][1] and ftypes[f][2] defines lower (1) and upper (2) extensions $e_t(f)$ and $e_u(f)$ required for the type $f$ field-arrays, besides extensions for communication. That is, for a subdomain of $[0, \sigma_{\theta}-1] \times \cdots \times [0, \sigma_{D-1}-1]$, the array for $f$ is at least as large as to have grid points of $[e_t(f), \sigma_{\theta}+e_u(f)-1] \times \cdots \times [e_t(f), \sigma_{D-1}+e_u(f)-1]$. Note that if the field-arrays of type $f$ do not need such non-communicational extensions, you should let $e_t(f) = e_u(f) = 0$.

ftypes[f][3] and ftypes[f][4] defines lower (3) and upper (4) extensions $e_l(f)$ and $e_u(f)$ for the broadcast of the type $f$ field-arrays. For example, for your electromagnetic field $eb[2][x][y]$ of type $f$ for a subdomain of $[0, \sigma_{x}-1] \times [0, \sigma_{y}-1] \times [0, \sigma_{z}-1]$, oh3_bcast_field() sends structured elements in the range\(^{10}\):

- from $eb[0][e_l(f)] [e_l(f)] [e_l(f)]$
- to $eb[0][\sigma_{x}+e_l(f)-1] [\sigma_{y}+e_l(f)-1] [\sigma_{z}+e_l(f)-1]$

Note that if the field-arrays of type $f$ are never broadcasted, you should let $e_l(f) = e_u(f) = 0$.

ftypes[f][5] and ftypes[f][6] defines lower (5) and upper (6) extensions $e_l(f)$ and $e_u(f)$ for the reduction of the type $f$ field-array. For example, for your current density array of type $f$ namely cd[2][x][y] having structured elements of three vector components for a subdomain of $[0, \sigma_{x}-1] \times [0, \sigma_{y}-1] \times [0, \sigma_{z}-1]$, oh3_allreduce_field() or oh3_reduce_field() performs the reduction of the elements in the range\(^{11}\):

- from $cd[0][e_l(f)] [e_l(f)] [e_l(f)]$
- to $cd[0][\sigma_{x}+e_u(f)-1] [\sigma_{y}+e_u(f)-1] [\sigma_{z}+e_u(f)-1]$

Note that if you will never perform reductions on the field-arrays of type $f$, you should let $e_l(f) = e_u(f) = 0$.

*cfields should be a pointer to an array of $C+1$ elements and its element cf\(\)ields\[c\] should has $f \in [0, F-1]$ to identify a field-array type for which a type of boundary communication identified by the integer $c \in [0, C-1]$ is defined, while c\(\)ypes[C] should be $-1$ (or less) to tell oh3_init() that you have $C$ types of boundary communications.

This array implies that a field-array may have two or more boundary communication types according to the timing of the communication, or no boundary communication may be taken for the field-array.

*c\(\)ypes should be a pointer to an array of $C \times B \times 2 \times 3$ to form c\(\)ypes[C][B][2][3]

conceptually. Its elements c\(\)ypes[c][b][w][s] = ($e_f$, $e_s$, $s$) defines downward ($w = 0$) or

\(^{10}\)Not the set of structured elements

\{\(eb[0][x][y][z]|x \in [e_l(f), \sigma_{x}+e_l(f)-1], y \in [e_l(f), \sigma_{y}+e_l(f)-1], z \in [e_l(f), \sigma_{z}+e_l(f)-1]\)\}

\(^{11}\)Not the set of structured elements

\{\(cd[0][x][y][z]|x \in [e_l(f), \sigma_{x}+e_u(f)-1], y \in [e_l(f), \sigma_{y}+e_u(f)-1], z \in [e_l(f), \sigma_{z}+e_u(f)-1]\)\}

46
upward ($w = 1$) boundary communication through the boundary of type $b \in [0, B-1]$ for a field-array $f = \text{cfields}[c]$ of the subdomain of $[0, \sigma_0-1] \times \ldots [0, \sigma_D-1]$ as follows (Figure 13).

- Downward ($w = 0$) communication along $d$-th dimensional axis is the pair of sending planes perpendicular to the axis to the lower neighbor and receiving the planes from the upper neighbor. The first plane to be sent has $d$-th dimensional coordinate $e_f$, while that to be received is at $\sigma_d + e_t$.

- Upward ($w = 1$) communication along $d$-th dimensional axis is the pair of sending planes perpendicular to the axis to the upper neighbor and receiving the planes from the lower neighbor. The first plane to be sent has $d$-th dimensional coordinate $\sigma_d + e_f$, while that to be received is at $e_t$.

Therefore, when you just need $s_l$ and $s_u$ planes at the lower and upper boundaries surrounding a subdomain, $e_f = e_t = 0$ and $s = s_u$ for downward communication, while $e_f = e_t = -s_l$ and $s = s_l$ for upward communication as shown in Figure 13(b). On the other hand, if you need these planes keeping those calculated by the local node for, e.g., the addition of current densities at boundaries, $e_t = e_f + s_u$ and $s = s_u$ for downward communication, while $e_f = e_t + s_l$ for upward communication, as shown in Figure 14.

Note that if no data is transferred by downward and/or upward type $c$ communication through a boundary of type $b$, the element $\text{ctors}_c[b][w][2]$, i.e., $s$, should be set to 0.

**fsizes** should be a double pointer to an array of $F \times D \times 2$ to form $\text{fsizes}[F][D][2]$ conceptually, or a pointer to NULL (not NULL itself) if you want the library to allocate the array and return the pointer to it through the argument. In both cases, its element $\text{fsizes}[f][d][\beta]$ will have $\phi^D_f(\beta = 0)$ or $\phi^D_f(\beta = 1)$ for the field-arrays of type $f$ to notify you that the field-arrays must have the size of $\phi^D_f(f) \times \epsilon$ for each of primary and secondary subdomains, where $\epsilon = \text{ftypes}[f][0]$. That is, if $D = 3$ and your field-array for electromagnetic field vectors $\text{eb}[2][\ldots]$ of struct named $\text{ebfield}$ has type $\text{feb}$, you have to allocate the array by the following.

```c
int (*fs)[3][2] = (*int(*)[3][2])(*fsizes);
int s[3] = (fs[feb][0][1] - fs[feb][0][0],
    fs[feb][1][1] - fs[feb][1][0],
    fs[feb][2][1] - fs[feb][2][0]);
int lext = fs[feb][0][0] + s[0] * (fs[feb][1][0] + s[1] * fs[feb][2][0]);
struct ebfield* eb0 = malloc(sizeof(struct ebfield) * s[0] * s[1] * s[2] * 2) - lext;
    eb[1] = eb[0] + s[0] * s[1] * s[2];
```

Note that the allocation above makes $\text{eb}[0]$ and $\text{eb}[1]$ points the origin of the subdomain at $(0, 0, 0)$ in its local integer coordinate system. Therefore, if you like to make $\text{eb}[1]$ point some other grid point, for example $(1, 2, 3)$, you have to modify $\text{lext}$ above as follows.

```c
int lext = (fs[feb][0][0] - 1) +
    s[0] * (fs[feb][1][0] - 2) + s[1] * (fs[feb][2][0] - 3));
```
The value of $\phi_d(f)$ and $\phi_u(f)$ are calculated by the followings to obtain the maximum extensions at lower and upper boundaries from $\text{ftypes}[]$, $\text{cfields}[]$ and $\text{ctypes}[]$, and the maximum size of each subdomain edge from $\text{sdoms}[]$. 

$$\Gamma(f) = \{ c | \text{cfields}[c] = f \}$$
$$\lambda(c, s) = \begin{cases} 
    c & s \neq 0 \\
    0 & s = 0
\end{cases}$$
$$s^+(b, c) = \text{ctypes}[c][b][0][2]$$
$$s^+(b, c) = \text{ctypes}[c][b][1][2]$$
$$e^+_{ij}(b, c) = \lambda(\text{ctypes}[c][b][0][0], s^+(b, c))$$
$$e^+_{ij}(b, c) = \lambda(\text{ctypes}[c][b][0][1], s^+(b, c))$$
$$e^+_{ij}(b, c) = \lambda(\text{ctypes}[c][b][1][0], s^+(b, c))$$
$$e^+_{ij}(b, c) = \lambda(\text{ctypes}[c][b][1][1], s^+(b, c))$$

$$e^2_{ij}(f) = \min_{b \in [0, B-1), c \in \Gamma(f)} \{(e^+_{ij}(b, c)) \cup \{e^+_{ij}(b, c)\}\}$$
$$e^2_{ij}(f) = \max_{b \in [0, B-1), c \in \Gamma(f)} \{(e^+_{ij}(b, c)) \cup \{e^+_{ij}(b, c)\}\}$$
$$\phi_d^2(f) = \min(e^2_{ij}(f), e_l(f), e^2_{ij}(f), e^r(f))$$
$$\phi_u^{\text{max}}(f) = \max(e^2_{ij}(f), e_u(f), e^2_{ij}(f), e^r(f))$$
$$\phi_u^{\text{max}}(f) = \max_{m \in [0, N-1]} \{\delta^2_u(m) - \delta^2_d(m)\}$$
$$\phi_u^{\text{max}}(f) = \phi_d^{\text{max}} + e_u^{\text{max}}(f)$$

For example, suppose $D = 2$, the subdomain decomposition is done as shown in Figure 9 with fully periodic boundaries, and you specify the followings for your electromagnetic field array $\text{eb}[]$ with field-array type identifier $\text{feb}$ and boundary communication type identifier $\text{ceb}$.

\begin{verbatim}
    ftypes[feb][0]=6;
    ftypes[feb][1]=0; ftypes[feb][2]=0;
    ftypes[feb][3]=0; ftypes[feb][4]=1;
    ftypes[feb][5]=0; ftypes[feb][6]=0;
    cfield[ceb]=feb;
    ctypes[ceb][0][0][0]=ctypes[ceb][0][0][1]=0;
    ctypes[ceb][0][0][2]=2;
    ctypes[ceb][0][1][0]=ctypes[ceb][0][1][1]=-1;
    ctypes[ceb][0][1][2]=1;
\end{verbatim}

Then you will have the followings in $\text{sizes}[]$ to allocate the array of six-element structures of $(1 + 8) \times (1 + 9) \times 2$.

\begin{verbatim}
    sizes[feb][0][0] = min(min(0, -1), 0, 0, 0) = -1
    sizes[feb][0][1] = 6 + max(max(2, 0), 0, 1, 0) = 6 + 2 = 8
    sizes[feb][1][0] = min(min(0, -1), 0, 0, 0) = -1
    sizes[feb][1][1] = 7 + max(max(2, 0), 0, 1, 0) = 7 + 2 = 9
\end{verbatim}
See Section 3.4.1 because the arguments above are perfectly equivalent to those of oh1_init().

### 3.6.2 oh13_init()

The function (subroutine) oh13_init() performs what oh3_init() does excluding the initialization of oh2_init() but including that of oh1_init(). More specifically, let \( I_1, I_2 \) and \( I_3 \) be the set of initializing operations performed in oh1_init(), oh2_init() and oh3_init() respectively, and thus \( I_1 \subset I_2 \subset I_3 \). The function oh13_init() performs \( I_3 - (I_2 - I_1) \) for those who want to have functions provided by level-3 library but to transfer and manage particles by themselves. Therefore oh13_init() does not allocate the large buffer for particle transfer. It also inhibits particle transfer operations in oh3_transbound() to make it almost equivalent to oh1_transbound() besides a few necessary operations for field-arrays.

The definition \( I_3 = (I_2 - I_1) \) of the initialization by oh13_init() is similarly applicable to its arguments. That is, its set of arguments is \( A_3 = (A_2 - A_1) \cup A_1 \) where \( A_k \) is the set of arguments of ohk_init(). Note that two arguments rcounts and scounts of oh1_init(), which is excluded from oh2_init() and thus also from oh3_init(), is in the set of oh13_init().

**Fortran Interface**

```fortran
subroutine oh13_init(sdid, nspec, maxfrac, mphgram, totalp, &
                    rcounts, scounts, mycomm, nbor, pcoord, &
                    sdoms, scoord, nbound, bcond, bounds, ftypes, &
                    cfields, ctypes, fsizes, &
                    stats, repiter, verbose)
  use oh_type
  implicit none
  integer,intent(out) :: sdid(2)
  integer,intent(in) :: nspec
  integer,intent(in) :: maxfrac
  integer,intent(inout) :: mphgram(:,:,:,)
  integer,intent(out) :: totalp(:,:)
  integer,intent(out) :: rcounts(:,:,:,)
  integer,intent(out) :: scounts(:,:,:,)
  type(oh_mycomm),intent(out) :: mycomm
  integer,intent(inout) :: nbor(OH_DIMENSION) ! for 3D codes.
  integer,intent(in) :: pcoord(OH_DIMENSION)
  integer,intent(inout) :: sdoms(:,:,:,)
  integer,intent(in) :: scoord(OH_DIMENSION)
  integer,intent(in) :: nbound
  integer,intent(in) :: bcond(OH_DIMENSION)
  integer,intent(inout) :: bounds(:,:)
  integer,intent(in) :: ftypes(:,:,)
  integer,intent(in) :: cfields(:)
  integer,intent(in) :: ctypes(:,:,,:,)
  integer,intent(out) :: fsizes(:,:,)
  integer,intent(in) :: stats
  integer,intent(in) :: repiter
  integer,intent(in) :: verbose
end subroutine
```
C Interface

```c
void oh13_init(int **sdid, int nspec, int maxfrac, int **nphgram,
                int **totalp, int **rcounts, int **scounts,
                void *mycomm, int **nbor, int *pcoord,
                int **sdoms, int *scoord, int nbound, int *bcond,
                int **bounds, int *ftypes, int *cfields, int *ctypes,
                int **fsizes,
                int stats, int repiter, int verbose);
```

sdid nspec maxfrac nphgram totalp rcounts scounts mycomm nbor pcoord

See Section 3.4.1 because the arguments above are perfectly equivalent to those of
`oh1_init()`.

sdoms scoord nbound bcond bounds ftypes cfields ctypes fsizes

See Section 3.6.1 because the arguments above are perfectly equivalent to those of
`oh3_init()`.

stats repiter verbose

See Section 3.4.1 because the arguments above are perfectly equivalent to those of
`oh1_init()`.

3.6.3 oh3_transbound()

If you initialize the library by `oh3_init()`, the function `oh3_transbound()` at first performs
the same operations as `oh2_transbound()` does; that is, examination of the balancing and
(re)building of helpand-helper configuration if necessary, followed by particle transfer. Oth-
ewise, i.e., if you have called `oh13_init()`, `oh3_transbound()` acts as `oh1_transbound()`
to make particle transfer schedule. Finally, in both cases, `oh3_transbound()` maintains
library’s internal data structures for field-arrays of the secondary subdomain, if helpand-
helper configuration has been (re)built. For this maintenance, the function refers to the
information given to `oh3_init()` but not the argument arrays themselves.
Since the arguments of \texttt{oh3\_transbound()} and its return value are perfectly equivalent to those of \texttt{oh1\_transbound()} (and \texttt{oh2\_transbound()}), see Section 3.4.2 for their definitions.

**Fortran Interface**

```fortran
integer function oh3_transbound(currmode, stats)
    implicit none
    integer,intent(in) :: currmode
    integer,intent(in) :: stats
end function
```

**C Interface**

```c
int oh3_transbound(int currmode, int stats);
```

### 3.6.4 \texttt{oh3\_map\_particle\_to\_neighbor()}  

The function \texttt{oh3\_map\_particle\_to\_neighbor()} returns the identifier of the subdomain in which the particle at given position will reside and to which the primary or secondary subdomain of the local node adjoins. Therefore, if the particle may be in a non-neighboring subdomain due to, for example, initial particle distribution, particle injection or particle warp, the relative function \texttt{oh3\_map\_particle\_to\_subdomain()} should be used.

Although the function is faster than \texttt{oh3\_map\_particle\_to\_subdomain()}, it is not good idea to use it to examine whether the particle is in the primary/secondary subdomain of the local node, because the calling cost is not negligible. That is, it is strongly recommended to do the examination by yourself and then call this function if you find the particle has gone.

This function has three instances with two, three and four arguments according to the dimension of the simulated space domain defined by $D = \text{OH\_DIMENSION}$.

**Fortran Interface**

```fortran
integer function oh3_map_particle_to_neighbor(x, ps)
    implicit none
    real*8,intent(inout) :: x
    integer,intent(in) :: ps
end function

integer function oh3_map_particle_to_neighbor(x, y, ps)
    implicit none
    real*8,intent(inout) :: x
    real*8,intent(inout) :: y
    integer,intent(in) :: ps
end function

integer function oh3_map_particle_to_neighbor(x, y, z, ps)
    implicit none
    real*8,intent(inout) :: x
    real*8,intent(inout) :: y
    real*8,intent(inout) :: z
    integer,intent(in) :: ps
end function
```
C Interface

```c
int oh3_map_particle_to_neighbor(double *x, int ps);
int oh3_map_particle_to_neighbor(double *x, double *y, int ps);
int oh3_map_particle_to_neighbor(double *x, double *y, double *z, int ps);
```

x, y, z (for Fortran)

*x, *y, *z (for C)

These three (if \( D = 3 \)) arguments should be the coordinates at which a particle is located in Fortran, or the pointers to the variables having the coordinates in C. In both cases, the actual argument variables may be updated as discussed later.

ps should be 0 for a primary particle, or 1 for a secondary particle.

**return value** is the identifier of the subdomain in which the particle will reside, or \(-1\) if such a subdomain is not found as discussed later.

The function at first examines whether the particle is in the primary (\( ps = 0 \)) or secondary (\( ps = 1 \)) subdomain of the local node and returns its identifier if the particle is in it, referring to the subdomain boundaries given by or set to the argument \texttt{sdoms} of \texttt{oh3_init()}. Otherwise, it assumes that the particle has moved into a subdomain adjoining to the primary/secondary subdomain and returns the identifier of the subdomain into which the particle has moved, referring to the neighboring information given by or set to the argument \texttt{nbor} of \texttt{oh3_init()}, or that in the helpand.

In the latter case of the boundary crossing, the periodic boundary condition of the whole space domain is taken care of by the function. Therefore, the coordinates given by \( x \), \( y \) and \( z \) should be raw ones without wraparound. Moreover, the actual argument variables are updated by the function if the particle has crossed a periodic boundary. For example, if the particle has crossed the periodic boundary plane perpendicular to \( x \)-axis, the actual argument variable \( x \) is updated as follows.

\[
x \leftarrow \begin{cases} 
x + (\Pi^u_x - \Pi^l_x) & x < \Pi^l_x \\
x - (\Pi^u_x - \Pi^l_x) & x \geq \Pi^u_x
\end{cases}
\]

On the other hand, if the particle has crossed a non-periodic boundary of the whole space domain, the function returns \(-1\) to indicate that the particle is out of bounds\(^{12}\). To examine the boundary condition, the function refers to the conditions given through the argument \texttt{bcond} or \texttt{bounds} of \texttt{oh3_init()}. The function also returns \(-1\) if the particle has moved into a non-existent neighbor, which may be defined by \texttt{nbor}.

### 3.6.5 oh3_map_particle_to_subdomain()

The function \texttt{oh3_map_particle_to_subdomain()} returns the identifier of the subdomain in which the particle at given position will reside. Unlike the relative function \texttt{oh3_map_particle_to_neighbor()}, this function can find the identifier of any subdomain and thus should be used for, e.g., initial particle distribution, particle injection, particle warp, and so on. Of course you may use this function always but have to remember that it is slower

---

\(^{12}\) The values in the actual argument variables are kept unless the particle has crossed two or more contacting space domain boundaries including periodic ones at once. More specifically, the function examines boundary crossing in the order of \( yz, xz \) and then \( xy \) planes if \( D = 3 \), and updates actual argument variables \( x, y \) and \( z \) in this order if the corresponding boundary planes are periodic.
than `oh3_map_particle_to_neighbor()` especially if you specify `sdoms` argument of `oh3_init()` by yourself.

This function has three instances with one, two and three arguments according to the dimension of the simulated space domain defined by $D = OH\_DIMENSION$.

**Fortran Interface**

```
integer function oh3_map_particle_to_subdomain(x)
    implicit none
    real*8,intent(in) :: x
end function

integer function oh3_map_particle_to_subdomain(x, y)
    implicit none
    real*8,intent(in) :: x
    real*8,intent(in) :: y
end function

integer function oh3_map_particle_to_subdomain(x, y, z)
    implicit none
    real*8,intent(in) :: x
    real*8,intent(in) :: y
    real*8,intent(in) :: z
end function
```

**C Interface**

```
int oh3_map_particle_to_subdomain(double x);
int oh3_map_particle_to_subdomain(double x, double y);
int oh3_map_particle_to_subdomain(double x, double y, double z);
```

`x, y` and `z` should be the coordinates at which a particle is located.

**Return value** is the identifier of the subdomain in which the particle will reside, or $-1$ if such a subdomain is not found as discussed later.

If you delegated the setting of `sdoms` array of `oh3_init()`, the function finds the subdomain by a simple calculation taking $O(1)$ time which should be, however, longer than that taken by `oh3_map_particle_to_neighbor()` due to an integer division. Therefore, it is not good idea to call this function to examine whether the particle is in the primary/secondary subdomain of the local node. That is, you should examine it by yourself and then, if the particle has gone outside, call this function. Also note that the calculation does not take care of the periodic boundary condition of the whole space domain, and thus you have to perform wraparound calculation before calling this function if necessary, or you will get the return value $-1$ to indicate that particle is out of bounds.

On the other hand, if you specify the array `sdoms` by yourself, this function *searches* the target subdomain. If your space domain is a cuboid (or a rectangler or a line segment) without any holes, the cost of search is $O(\log N)$. Otherwise, for a complicatedly shaped domain, the cost could be $O(N)$ although the function does its best to reduce it to $O(\log N)$. The search may fail if there is no subdomain including the given particle coordinates due to, for example, going outside the whole space domain, dropping into a hole, and so on, to make the function return $-1$.  

53
3.6.6 oh3_bcast_field()

The function (subroutine) oh3_bcast_field() performs red-black broadcast communications of a field-array whose type is specified by its argument ftype in the families the local node belongs to. The argument pfld specifies the field-array to be broadcasted in the primary family, while sfld is for the data to be broadcasted in the secondary family. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary broadcast if it is a leaf and the secondary one if it is the root. It is neither necessary to specify the data count because it is calculated by the library, nor to give MPI data-type to the function because MPI_DOUBLE_PRECISION for Fortran or MPI_DOUBLE for C is assumed.\footnote{Therefore, your field-arrays should have elements only of double precision floating point data or structures only of them.}

Fortran Interface

\begin{verbatim}
subroutine oh3_bcast_field(pfld, sfld, ftype)
  implicit none
  real*8,intent(in) :: pfld
  real*8,intent(out) :: sfld
  integer,intent(in) :: ftype
end subroutine
\end{verbatim}

C Interface

\begin{verbatim}
void oh3_bcast_field(void *pfld, void *sfld, int ftype);
\end{verbatim}

pfld should be (the pointer to) the first field-array element at the origin of the primary subdomain. The contents of the field-array are broadcasted from the local node to its helpers in its primary family.

sfld should be (the pointer to) the first field-array element at the origin of the secondary subdomain. The broadcasted data in the secondary family is received to the field-array.

ftype should be the identifier to specify the type of the field-array.

For example, to broadcast your electromagnetic field-array \texttt{eb(6,:,:,:,2)} of type \texttt{feb}, you can simply do the following in your Fortran code providing the origins are \texttt{eb(:,0,0,0,:)}.

\begin{verbatim}
call oh3_bcast_field(eb(1,0,0,0,1),eb(1,0,0,0,2),feb)
\end{verbatim}

As for C field-array of \texttt{struct} whose origins are pointed by \texttt{eb[0]} and \texttt{eb[1]}, what you have to do is simply the following.

\begin{verbatim}
oh3_bcast_field(eb[0],eb[1],feb);
\end{verbatim}

In order to make the interfaces simple as shown above, the function refers to \(e^p_f(f)\) and \(e^b_f(f)\) for \(f = ftype\) given in the argument ftypes of oh3_init(), and the size of primary/secondary subdomain given in sdoms and that of the field-array itself set to fsizes. Note that the elements to be broadcasted are not only in the subarray defined by \(e^p_f(f)\) and \(e^b_f(f)\) but also some of outside the subarray as shown in Figure 12 in Section 3.6.1 for the sake of efficiency. This overrun should not be harmful to the logical correctness of the simulation.
3.6.7 oh3_allreduce_field()

The function (subroutine) oh3_allreduce_field() performs red-black all-reduce summation of a field-array whose type is specified by its argument ftype in the families the local node belongs to. The argument pfld specifies the field-array to be reduced in the primary family, while sfld is for the data to be reduced in the secondary family. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary reduction if it is a leaf and the secondary one if it is the root. It is neither necessary to specify the data count because it is calculated by the library, to give MPI data-type to the function because MPI_DOUBLE_PRECISION for Fortran or MPI_DOUBLE is assumed, nor to tells it how the reduction is done because MPI_SUM is assumed\textsuperscript{14}.

**Fortran Interface**

```fortran
subroutine oh3_allreduce_field(pfld, sfld, ftype)
  implicit none
  real*8,intent(inout) :: pfld
  real*8,intent(inout) :: sfld
  integer,intent(in) :: ftype
end subroutine
```

**C Interface**

```c
void oh3_allreduce_field(void *pfld, void *sfld, int ftype);
```

pfld should be (the pointer to) the first field-array element at the origin of the primary subdomain. The contents of the field-array are replaced with the sum in the primary family.

sfld should be (the pointer to) the first field-array element at the origin of the secondary subdomain. The contents of the field-array are replaced with the sum in the secondary family.

ftype should be the identifier to specify the type of the field-array.

For example, to have the sum of your current density field-array cd(3,:,:,:,2) of type fcd, you can simply do the following in your Fortran code providing the origins are cd(:,0,0,0,:).

```fortran
call oh3_allreduce_field(cd(1,0,0,0,1),cd(1,0,0,0,2),fcd)
```

As for C field-array of struct whose origins are pointed by cd[0] and cd[1], what you have to do is simply the following.

```c
oh3_allreduce_field(cd[0],cd[1],fcd);
```

In order to make the interfaces simple as shown above, the function refers to e_r(f) and e_l(f) for \( f = \text{ftype} \) given in the argument ftypes of oh3_init(), and the size of primary/secondary subdomain given in sdoms and that of the field-array itself set to fsizes. Note that the elements to be reduced are not only in the subarray defined by \( e_r(f) \) and \( e_l(f) \) but also some of outside the subarray as shown in Figure 12 in Section 3.6.1 for the sake of efficiency. This overrun should not be harmful to the logical correctness of the simulation.\footnote{Therefore, the function cannot be used for any other reductions than summing up.}
3.6.8 oh3_reduce_field()

The function (subroutine) oh3_reduce_field() performs red-black one-way counterpart of the function oh3_allreduce_field().

Fortran Interface

```fortran
subroutine oh3_reduce_field(pfld, sfld, ftype)
   implicit none
   real*8,intent(inout) :: pfld
   real*8,intent(in) :: sfld
   integer,intent(in) :: ftype
end subroutine
```

C Interface

```c
void oh3_reduce_field(void *pfld, void *sfld, int ftype);
```

pfld should be (the pointer to) the first field-array element at the origin of the primary subdomain. The contents of the field-array are replaced with the sum in the primary family.

sfld should be (the pointer to) the first field-array element at the origin of the secondary subdomain. The contents of the field-array remain unchanged.

ftype should be the identifier to specify the type of the field-array.

3.6.9 oh3_exchange_borders()

The function (subroutine) oh3_exchange_borders() exchanges boundary planes of a field-array between adjacent primary subdomains. Then, if specified to do, the boundary planes are broadcasted from the local node to its helpers.

Fortran Interface

```fortran
subroutine oh3_exchange_borders(pfld, sfld, ctype, bcast)
   implicit none
   real*8,intent(inout) :: pfld
   real*8,intent(out) :: sfld
   integer,intent(in) :: ctype
   integer,intent(in) :: bcast
end subroutine
```

C Interface

```c
void oh3_reduce_field(void *pfld, void *sfld, int ctype, int bcast);
```

pfld should be (the pointer to) the first field-array element at the origin of the primary subdomain. The boundary planes (or line segments) of the field-array are sent/received to/from the nodes which are responsible for the subdomains adjoining to the primary subdomain of the local node as their primary ones.
sfld should be (the pointer to) the first field-array element at the origin of the secondary subdommain. The boundary planes of the field-array are replaced with that in the helpand of the local node, if bcast is non-zero and we are in secondary mode.

c-type should be the identifier to specify the type of the field-array communicaion, which is an index of ctypes of oh3_init().

bcast should be non-zero to broadcast obtained boundary planes to the helpers. If it is 0, only the boundary exchange of the primary subdomain is performed. Note that if we are in primary mode, the broadcast is not performed even if bcast \neq 0.

For example, you can simply do the following in your Fortran code to exchange boundary data of your electromagnetic field-array eb(6,:,:,:,2) of communication type ceb, providing the origins are eb(:,0,0,0,:) and you do not want to broadcast the received boundary planes.

```fortran
  call oh3_exchange_borders(eb(1,0,0,0,1),eb(1,0,0,0,2),ceb,0)
```

As for C field-array of struct whose origins are pointed by eb[0] and eb[1], what you have to do is simply the following.

```c
  oh3_exchange_borders(eb[0],eb[1],ceb,0);
```

By these simple statements, you can achieve fairly complicated communications as shown in Figure 13 of Sectin 3.6.1 because oh3_exchange_borders() takes care of various matters. First, it of course follows the specifications of the number of planes and their sources and destinations in the field-array given through the argument ctypes of oh3_init(). The specifications are also used to determine the size of a plane depending on the axis along which a communication is taken place. That is, the function enlarges the planes to be exchanged as it proceeds the communication from along x-axis then y and to z-axis, so that the local node obtains boundary data not only from the subdomains contacted with planes but also with edges and vertices as shown in Figure 13. Finally, to have the shape of the set of planes to be transferred and to represent them with a derivative data type of MPI, the function consults the size of primary/secondary subdomain given in sdoms and that of the field-array itself set to fsizes.

The finely designed boundary communication above is especially helpful for more complicated communications required to have the sum of current densities of a grid point around a vertices connecting subdomains. As shown in Figure 14 of Section 3.6.1, you can have 2^D partial sums calculated by 2^D families by a simple definition in ctypes and the following simple call in Fortran, providing your current density field-array is cd(3,:,:,:,2) and its type is ccd.

```fortran
  call oh3_exchange_borders(cd(1,0,0,0,1),cd(1,0,0,0,2),ccd,1)
```

Note that the boundary planes obtained by the communication between adjoined primary subdomains are broadcasted to the helpers of the local node if necessary in the example above. The C counterpart of the example is also simple as follows.

```c
  oh3_exchange_borders(cd[0],cd[1],ccd,1);
```
3.7 Particle Injection and Removal

As discussed in Section 3.5.5, level-2 library provides you with a function to dynamically inject a particle, `oh2_inject_particle()`. This section revisits this issue and also discusses its counterpart, particle removal.

If you use level-1 library only, what you need to do on injecting and/or removing particles is to maintain `nphgram` correctly as far as the library concerns. Since the function `oh1_transbound()` will not be surprised at a sudden apparition of a particle into any subdomain and any node, you may freely increase an element of `nphgram` to notify the library of the particle injection. This unusual increase of `nphgram` elements, however, may cost if particles are injected into a node which is not responsible for the subdomain to which the particles have appeared or for that adjoining the subdomain. That is, `oh1_transbound()` needs some global communications to make the particle transfer schedule, which are unnecessary on usual boundary crossing transfers. On the other hand, decreasing elements of `nphgram` to remove particles is no problem in terms of both logical correctness and performance of `oh1_transbound()`.

An important caution on the play with `nphgram` is that `oh1_transbound()` is only aware of the load balancing of particles whose populations in subdomains are reported in `nphgram`, of course. This means that if you have a stock of inactive particles in your particle buffer from which you pick particles to be injected and into which you fling removed particles, your buffer could overflow because `oh1_transbound()` does not know anything about the stock. Therefore, the stock should be sufficiently small, say up to some hundred thousands. Note that particle recycling without stock, i.e., injecting a particle only when another particle is removed by overwriting particle data, should cause no problem.

A way to avoid the overflow of the stock, especially when the stock is significantly large, is to include the number of particles in the stock into `nphgram` making them pretend to reside in a subdomain. This works well with respect to the balancing of required memory space but might cause severe imbalance of computation, because `oh1_transbound()` does not know that particles in the stock are inactive. Moreover, since `oh1_transbound()` may decide to throw particles in the stock away to other nodes, the node could find it has no particles to recycle in the stock on injection.

On the other hand, an injection by `oh2_inject_particle()` is not only as easy as just increasing `nphgram` but also consistent with other library functions especially with `oh2_transbound()` (and thus `oh3_transbound()` usually), which recognizes the particle, the subdomain into which it is injected, and the memory location at which it is stored. That is, `oh2_transbound()` automatically picks injected particles from the bottom of `pbuf` and places them into appropriate position in `pbuf` or transfers them to appropriate nodes which are responsible for the subdomains they reside. What you need to take care of is that you have to reserve some space (not a stock) in `pbuf` large enough to inject particles in a simulation time step. If the space is too large for a node due to a significantly large number of potential injections, you can limit the space to a reasonable size and let the node having too many particles to be injected push overflown ones to other nodes. A simple solution to do it is to repeat `oh2_transbound()` and an all-reduce communication to confirm the completion of all particle injections, because it is assured that the space for injection is emptied each time `oh2_transbound()` is executed.

Particle removal can be implemented more easily with level-2 or level-3 library. What you need to do is to set `nid` element of the particle in problem to be −1, excluding it from counting particles for `nphgram`. Then `oh2_transbound()` will remove the particles

15 Unless the total of `nphgram` reaches or exceeds $2^{31} - 1$.
16 Or skipping the increment of `nphgram` element for the particle to be removed.
reclaiming the space for them.

The last issue on particle injection and removal is the identification of particles. In the default definition of the Fortran structured type \texttt{oh\_particle} and C \texttt{struct} named \texttt{S\_particle}, each particle has its identifier in \texttt{pid} element. Since this element is a 64-bit integer, the space for the identification number is large enough for local numbering without reclamation. For example, a node \( n \) may give a number \( kN + n \) to the \( k \)-th particle created by the node. Since \( 2^{64} \) should be much larger than \( N \), the identification space is hardly exhosted. For example, even if \( N = 2^{20} \) and each node injects (and removes) \( 2^{20} \) particles in each simulation step in addition to its initial accommodation of \( 2^{30} \) particles, it will take about 16.8 million time steps or, even if your simulator has an excellent per-node performance of 10 million particles per second\footnote{The per-node performance of our simulator reported in [1] is 2.55 million particle per second.}, 1.68 billion seconds or 53 years.

### 3.8 Statistics

Level-1 library provides you with the functions to collect, process and report two types of statistics data of timings and particle transfers. The timing statistics data is obtained by measuring the execution time of intervals in your program including the library functions. Since each interval is identified by a \textit{key} being a non-negative unique integer, you have to define the set of keys for the intervals which you want to measure together with strings printed on the report, by modifying the C header file \texttt{oh\_stats.h} as discussed in Section 3.8.1. Then, after calling \texttt{oh1\_init()}, or one of its higher level counterparts \texttt{oh2\_init()} and \texttt{oh3\_init()}, giving it fundamental parameters for statistics as discussed in Section 3.8.2, you may call the following functions to collect, process and report statistics data as explained in Section 3.8.3, 3.8.4, 3.8.5 and 3.8.6.

\texttt{oh1\_init\_stats()} initializes internal data structures for statistics and starts the execution time measurement of the first interval.

\texttt{oh1\_stats\_time()} finishes the execution time measurement of the last interval, and starts that of the next interval.

\texttt{oh1\_show\_stats()} gathers timing and particle transfer statistics data measured in a simulation step and, if specified, reports a subtotal for recent steps.

\texttt{oh1\_print\_stats()} reports the grand total of statistics data.

#### 3.8.1 Timing Statistics Keys and Header File \texttt{oh\_stats.h}

You can measure the execution time of an interval in your program by calling \texttt{oh1\_stats\_time()} giving it a key to identify the interval. Since the key, a non-negative integer, should be unique to the interval and should be associated to a character string printed on the report together with the statistics of the measured timing, the library provides you with a C header file named \texttt{oh\_stats.h}, which can be included from Fortran codes too, to assure the uniqueness and the association with the string easily.

The file consists of two parts and the default definition given by the first part is as follows.

\begin{verbatim}
#define STATS_TRANSBOUND 0
#define STATS_TRY_STABLE (STATS_TRANSBOUND + 1)
#define STATS_REBALANCE (STATS_TRY_STABLE + 1)
#define STATS_REB_COMM (STATS_REBALANCE + 1)
\end{verbatim}
The code above `define`'s the following six keys to measure the execution times in `oh1_transbound()` and/or its higher level counterparts and a special key `STATS_TIMINGS` to have the number of keys.

`STATS_TRANSBOUND` is for the interval to examine if the execution mode in the next step is primary.

`STATS_TRY_STABLE` is for the interval to examine if the helpand-helper configuration can be kept in the next step.

`STATS_REBALANCE` is for the interval to (re)build a new helpand-helper configuration.

`STATS_REB_COMM` is for the interval to create family communicators for the newly built helpand-helper configuration.

`STATS_TB_MOVE` is for the interval in `oh2_transbound()` or `oh3_transbound()` to move particles in `pbuf`.

`STATS_TB_COMM` is for the interval in `oh2_transbound()` or `oh3_transbound()` to transfer particles among nodes.

On adding your own keys, it is recommended to follow the convention shown in the file. That is, defining a key by:

```
#define ⟨new key⟩ (⟨last key⟩ + 1)
```

will assure the uniqueness and continuity of keys. For example, to add three keys namely `STATS_PARTICLE_PUSHING`, `STATS_CURRENT_SCATTERING` and `STATS_FIELD_SOLVING` for the intervals of particle pushing, current scattering and field solving in your main loop, replacing the first line for `STATS_TRANSBOUND` with the followings is safe and correct.

```
#define STATS_PARTICLE_PUSHING 0
#define STATS_CURRENT_SCATTERING (STATS_PARTICLE_PUSHING + 1)
#define STATS_FIELD_SOLVING (STATS_CURRENT_SCATTERING + 1)
#define STATS_TRANSBOUND (STATS_FIELD_SOLVING + 1)
```

Note that you must not remove any definitions given in the original `oh_stats.h`, or you cannot compile the library correctly.

The second part of the file defines the character strings for keys as follows.

```
#define OH_DEFINE_STATS
static char *StatsTimeStrings[2*STATS_TIMINGS] = {
  "transbound", "",
  "try_stable", "",
  "rebalance", "",
  "reb comm create", "",
  "part move[pri]", "part move[sec]",
  "part comm[pri]", "part comm[sec]",
};
#endif
```
In the code above, `#ifdef/#endif` construct is to protect your code from erroneous compilation especially if your code is in Fortran. That is, `OH_DEFINE_STATS` is defined only in the library source files and thus the compiler for your own codes will skip the part which cannot be parsed as a Fortran code.

The important part in the code is the sequence of the string pairs, one pair for each line. The pairs correspond to keys in the same order and each pair gives a short explanation of the pair of intervals, one for primary particles/subdomains and the other for secondary ones, identified by the corresponding key. That is, if your interval is executed twice as a primary execution and a secondary execution, the first and second strings are used as the titles of two executions separately. Otherwise, or if you measure two executions as a whole, defining the first string and letting the second be empty string are necessary and sufficient.

For example, adding the following three lines just before the line having "transbound" is what you need to do for the three keys exemplified above, providing you want to measure primary and secondary executions of each interval separately.

```
"particle pushing[pri]",  "particle pushing[sec]",
"current scattering[pri]",  "current scattering[sec]",
"field solving[pri]",      "field solving[sec]",
```

Remember that a title can be arbitrarily long but that of 30 characters or longer will cause an ungly line in the report.

### 3.8.2 Arguments of oh1_init() for Statistics

As shown in Section 3.4.1, the function `oh1_init()` and its higher level counterparts have the following two arguments to control statistics operations.

- `stats` activates or inactivates statistics operations as follows.
  - If `stats = 0`, statistics operations are inactivated and thus the functions discussed in the following sections do nothing.
  - if `stats = 1`, statistics operations are activated but only the grand total is reported by `oh1_print_stats()`.
  - if `stats = 2`, statistics operations are activated and `oh1_show_stats()` will report subtotal when it is given the simulation step count divisible by the argument `repiter` of `oh1_init()`.

Note that `oh1_transbound()` and its higher level counterparts also have an argument `stats` to control the statistics collection in the function temporarily overriding what `stats` of `oh1_init()` specifies. That is, statistics collection in `oh1_transbound()` is inactivated if its `stats` is 0 regardless `stats` of `oh1_init()`, while non-zero means that statistics collection follows what `stats` of `oh1_init()` specifies. This feature is useful to exclude statistics data in, for example, initialization process.

- `repiter` defines the frequency of subtotal reporting by `oh1_show_stats()`. That is, if `stats = 2`, it defines the gap of periodical reporting by `oh1_show_stats()`.

---

18 Or some other value excluding 0 and 2.
3.8.3 oh1_init_stats()

The function (subroutine) oh1_init_stats() initializes internal data structures for statistics, and starts first interval of timing measurement, if stats of oh1_init() is not zero. The other statistics function must be called after oh1_init_stats() is called.

**Fortran Interface**

```fortran
subroutine oh1_init_stats(key, ps)
implicit none
integer,intent(in) :: key
integer,intent(in) :: ps
end subroutine
```

**C Interface**

```c
void oh1_init_stats(int key, int ps);
```

key is the key of the first interval whose execution time is measured. If you do not want to include the first interval in the timing statistics, give this argument the special key STATS_TIMINGS.

ps indicates whether the first interval is for primary execution (0) or secondary execution(1).

3.8.4 oh1_stats_time()

The function (subroutine) oh1_stats_time() finishes the last interval of timing measurement and starts next one, if stats of oh1_init() is not zero.

**Fortran Interface**

```fortran
subroutine oh1_stats_time(key, ps)
implicit none
integer,intent(in) :: key
integer,intent(in) :: ps
end subroutine
```

**C Interface**

```c
void oh1_stats_time(int key, int ps);
```

key is the key of the interval to start for execution time measurement. If you want only to finish the last interval, give this argument the special key STATS_TIMINGS.

ps indicates whether the next interval is for primary execution (0) or secondary execution(1).
3.8.5 oh1_show_stats()

The function (subroutine) `oh1_show_stats()` performs the following statistics operations if `stats` of `oh1_init()` non-zero.

- Finish the last interval of timing measurement.
- Gather statistics data measured since the last call of this function or the call of `oh1_init_stats()`.
- Update grand total statistics and, if `stats` of `oh1_init()` is 2, subtotal statistics.
- Print subtotal statistics as `oh1_print_stats()` does, if `stats` of `oh1_init()` is 2 and `step` argument of this function is divisible by `repiter` of `oh1_init()`.
- Start a new interval whose execution time is excluded from timing statistics.

It is expected to call this function every simulation step so that it collect statistics data for each step.

Fortran Interface
```
subroutine oh1_show_stats(step, currmode)
    implicit none
    integer, intent(in) :: step
    integer, intent(in) :: currmode
end subroutine
```

C Interface
```
void oh1_show_stats(int step, int currmode);
```

`step` is the simulation step count to control periodical statistics reporting. If `stats` of `oh1_init()` is 2 and `step` is divisible by `repiter` of `oh1_init()`, subtotal statistics is reported.

`currmode` indicates whether the current execution mode is primary (0) or secondary (1). This value should be corresponding to the return value of the last call of `oh1_transbound()` or its higher level counterparts.

3.8.6 oh1_print_stats()

The function (subroutine) `oh1_print_stats()` report the grand total (so far) of statistics through standard output in the following format. The first part of the report is for execution time of each interval as follows.

```
# Execution Times (sec)
particle pushing[pri] = 0.024 / 2.297 / 1.015 / 1824925.604
particle pushing[sec] = 0.077 / 2.440 / 1.564 / 2135827.627
current scattering[pri] = 0.011 / 1.223 / 0.422 / 736536.722
current scattering[sec] = 0.032 / 1.332 / 0.836 / 1296109.407
field solving[pri] = 0.003 / 0.089 / 0.011 / 27344.603
field solving[sec] = 0.003 / 0.053 / 0.012 / 19633.007
transbound = 0.004 / 0.837 / 0.222 / 364201.720
```

63
Each column of the table above shows the followings of each interval.

- Column-1: title of the interval.
- Column-2: minimum execution time of the interval.
- Column-3: maximum execution time of the interval.
- Column-4: average execution time of the interval.
- Column-5: sum of execution times of the interval.

Note that the minimum, maximum, average and sum are over all occasions of each interval in all nodes and all simulation time steps.

Then the second part reports the statistics of particle transfer as follows.

## Particle Movements

<table>
<thead>
<tr>
<th>Type</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2p transfer[pri,min]</td>
<td>235 / 4272367</td>
<td>7368 / 47153707</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2p transfer[pri,max]</td>
<td>1891 / 8054375</td>
<td>14809 / 95416324</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2p transfer[pri,ave]</td>
<td>441 / 6194818</td>
<td>12796 / 81894514</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get[pri,min]</td>
<td>0 / 589</td>
<td>3 / 19210</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get[pri,max]</td>
<td>6511 / 8054765</td>
<td>22971 / 147011962</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put[pri,min]</td>
<td>0 / 984</td>
<td>5 / 29490</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put[pri,max]</td>
<td>6209 / 8054375</td>
<td>16387 / 104877429</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put&amp;get[pri,ave]</td>
<td>13 / 31464</td>
<td>90 / 574318</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2p transfer[sec,min]</td>
<td>1 / 656</td>
<td>2 / 10488</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2p transfer[sec,max]</td>
<td>2198 / 6034178</td>
<td>22907 / 146602986</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2p transfer[sec,ave]</td>
<td>31 / 1393581</td>
<td>2748 / 17587875</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get[sec,min]</td>
<td>0 / 289</td>
<td>2 / 10021</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get[sec,max]</td>
<td>3577 / 8387296</td>
<td>51544 / 329883298</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put[sec,min]</td>
<td>0 / 1476</td>
<td>4 / 24108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put[sec,max]</td>
<td>3809 / 9732486</td>
<td>47848 / 306224944</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put&amp;get[sec,ave]</td>
<td>118 / 1812744</td>
<td>3473 / 22225683</td>
<td></td>
<td></td>
</tr>
<tr>
<td>transition to pri</td>
<td>1594 / 0</td>
<td>1 / 1595</td>
<td></td>
<td></td>
</tr>
<tr>
<td>transition to sec</td>
<td>1 / 4782</td>
<td>22 / 4805</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The rows above except for the last two are for the following particle transfers which are scheduled in one execution of `oh1_transbound()` or are actually performed in one execution of `oh2_transbound()` or `oh3_transbound()`.

p2p transfer[] shows the number of transferred particles between a pair of nodes. The minimum, maximum and average are calculated over all pairs such that at least one particle is transferred between each pair nodes.
**get[]** shows the number of particles a node received. The minimum and maximum are calculated over all nodes including those received nothing.

**put[]** shows the number of particles a node sent. The minimum and maximum are calculated over all nodes including those sent nothing.

**put&get[]** shows the average number of particles a node received (or sent). The average are calculated over all nodes including those received nothing.

Note that the categorization of primary (**pri**) and secondary (**sec**) particles is based on the viewpoint of receivers. Also note that the columns from Column-2 to Column-5 of these rows are for the minimum, maximum, average and sum which are calculated over all simulation time steps.

On the other hand, the last two rows shows number of transitions to primary and secondary modes. In these rows, Column-2 and Column-3 are for the number of transitions from primary and secondary modes respectively. Column-4 of the transition to primary is the number of primary to primary transition at which non-neighboring particle transfers are taken, while that of to secondary means the number of secondary to secondary with rebuilding of helpand-helper configuration. Finally Column-5 of both rows is the total number of transition to primary or secondary mode.

The function **oh1_show_stats()** also reports the statistics if **stats** and **repiter** of **oh1_init()** and **step** argument of the function satisfy the reporting condition, but the numbers shown in columns of the minimum and others are calculated over the recent **repiter** steps.

**Fortran Interface**

```fortran
subroutine oh1_print_stats(nstep)
  implicit none
  integer,intent(in) :: nstep
end subroutine
```

**C Interface**

```c
void oh1_print_stats(int nstep);
```

**nstep** is the total simulation step count to calculate the average numbers in Column-4.

### 3.9 Verbose Messaging

Although the application of the OhHelp library to your PIC simulator is fairly simple and straightforward, it should be hard to compose a bug-free program instantly. Therefore, you will want to investigate what is going on in your program including the functions in the library when you encounter a problem.

Verbose messaging provided by the library is a fundamental mean for the investigation. You can activate or inactivate the verbose messaging in library functions by giving one of the followings to the argument **verbose** of **oh1_init()** or its higher level counterparts.

- **verbose = 0** inactivates verbose messaging and thus makes library functions execute silently.
- **verbose = 1** activates verbose messaging to have fundamental reports from library functions.
• verbose = 2 activates more verbose messaging than the case of 1 to capture some details of the events happening in library functions.

• verbose = 3 is similar to 2 but you will have messages from all nodes with their identifier (MPI rank).

If activated, messages are printed to standard output with a common header “*Starting” optionally followed by a node identifier surrounded by brackets.

In addition, you may have your own verbose messaging to be controlled by verbose of oh1_init() by calling the following function oh1_verbose().

**Fortran Interface**

```
subroutine oh1_verbose(message)
  implicit none
  character(*),intent(in) :: message
end subroutine
```

**C Interface**

```
void oh1_verbose(char *message);
```

*message* is a character string to be printed following the header. Since it should be null-terminated, you have to remember that a Fortran string constant, say ‘hello’ does not have the terminator and thus you have to explicitly give a null code by ‘hello\0’.

Note that your message is assumed fundamental and thus will be printed if verbose is 1 or larger. Also note that oh1_verbose() has MPI_Barrier() in it and thus it should be called from all nodes to avoid deadlock. For example;

```
if (sdid(2).ge.0) then
  oh1_verbatim('secondary particle push\0')
  call particle_push(...)  
end if
```

will cause deadlock because the root node of helpand-helper tree will not call oh1_verbatim() while others do. Therefore, the code above should be modified as follows.

```
if (currmode.ne.0)
  oh1_verbatim('secondary particle push\0')
  if (sdid(2).ge.0) call particle_push(...)  
end if
```

### 3.10 Aliases of Functions

As shown in previous sections, all library functions have one of prefixes ‘oh1’, ‘oh2’ or ‘oh3’ to show the library layer they belong to. Although this naming makes it clear that in order to use a function, say oh2_inject_particle(), you have to incorporate level-2 or level-3 library, it will be tiresome to remember the layer number which a function belongs to especially when you use (almost) everything provided by the layer you chose and by lower ones.
Table 1: Aliases of Library Functions

<table>
<thead>
<tr>
<th>layer</th>
<th>alias</th>
<th>autonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>oh_broadcast()</td>
<td>oh1_broadcast()</td>
</tr>
<tr>
<td></td>
<td>oh_all_reduce()</td>
<td>oh1_all_reduce()</td>
</tr>
<tr>
<td></td>
<td>oh_reduce()</td>
<td>oh1_reduce()</td>
</tr>
<tr>
<td></td>
<td>oh_init_stats()</td>
<td>oh1_init_stats()</td>
</tr>
<tr>
<td></td>
<td>oh_stats_time()</td>
<td>oh1_stats_time()</td>
</tr>
<tr>
<td></td>
<td>oh_show_stats()</td>
<td>oh1_show_stats()</td>
</tr>
<tr>
<td></td>
<td>oh_print_stats()</td>
<td>oh1_print_stats()</td>
</tr>
<tr>
<td></td>
<td>oh_verbose()</td>
<td>oh1_verbose()</td>
</tr>
<tr>
<td>1</td>
<td>oh_init()</td>
<td>oh1_init()</td>
</tr>
<tr>
<td></td>
<td>oh_transbound()</td>
<td>oh1_transbound()</td>
</tr>
<tr>
<td>2 or 3</td>
<td>oh_max_local_particles()</td>
<td>oh2_max_local_particles()</td>
</tr>
<tr>
<td></td>
<td>oh_inject_particle()</td>
<td>oh2_inject_particle()</td>
</tr>
<tr>
<td>2</td>
<td>oh_init()</td>
<td>oh2_init()</td>
</tr>
<tr>
<td></td>
<td>oh_transbound()</td>
<td>oh2_transbound()</td>
</tr>
<tr>
<td>3</td>
<td>oh_init()</td>
<td>oh3_init()</td>
</tr>
<tr>
<td></td>
<td>oh_transbound()</td>
<td>oh3_transbound()</td>
</tr>
<tr>
<td></td>
<td>oh_map_particle_to_neighbor()</td>
<td>oh3_map_particle_to_neighbor()</td>
</tr>
<tr>
<td></td>
<td>oh_map_particle_to_subdomain()</td>
<td>oh3_map_particle_to_subdomain()</td>
</tr>
<tr>
<td></td>
<td>oh_bcast_field()</td>
<td>oh3_bcast_field()</td>
</tr>
<tr>
<td></td>
<td>oh_reduce_field()</td>
<td>oh3_reduce_field()</td>
</tr>
<tr>
<td></td>
<td>oh_allreduce_field()</td>
<td>oh3_allreduce_field()</td>
</tr>
<tr>
<td></td>
<td>oh_exchangeBorders()</td>
<td>oh3_exchange_borders()</td>
</tr>
</tbody>
</table>

Therefore, the library has special header files ohhelp.f.h for Fortran and ohhelp.c.h for C to give API function aliases which just have a common prefix ‘oh’. To use these files, you have to define a constant OH_LIB_LEVEL as the number of the layer you choose, i.e., 1, 2 or 3, or edit the line defining that it is 3 near by the top of the files. Then you have the aliases shown in Table 1 according to the layer number you chose.

Note that both header files #include’s a header file oh_dim.h, and ohhelp.c.h does the followings in addition to aliasing.

- #include the standard MPI header file mpi.h.
- Declares prototypes of library functions in use according to the layer you chose.
- Define struct named S.mycommc.
- #include the header file oh.part.h to define struct named S.particle if you choose level-2 or higher.

Also note that the function oh13Init() does not have any aliases.

3.11 Sample Code

This section gives examples of application of the level-3 OhHelp library to tiny 3-dimensional PIC simulators coded in Fortran and C. The main loop of these codes consists of calls of the following subroutines/functions, besides library functionalities.
particle_push() does what its name implies. The acceleration vector of each particle is calculated by a subroutine/function named lorentz() whose code is outside the scope of this document.

current_scatter() also does what its name indicates. The contribution of each particle to the current densities at grid points surrounding it is calculated by an out-of-scope subroutine/function named scatter().

add_boundary_current() calculates current density vectors of the grid points in boundary planes of a subdomain adding those obtained from neighboring subdomains to those calculated by the family members of the local one. This calls add_boundary_curr() for each boundary.

field_solve_e() is the first half of a leapfrog field solver to update electric field vectors. The rotation of magnetic field $\nabla \times B$ for the electric field vector of each grid point is calculated by an out-of-scope subroutine/function named rotate_b().

field_solve_b() is the second half of a leapfrog field solver to update magnetic field vectors. Similar to its electric counterpart, $\nabla \times E$ is calculated by an out-of-scope subroutine/function named rotate_e().

In addition to them, it is assumed that we have two out-of-scope subroutines/functions for initialization, namely initialize_eb() for electromagnetic field and initialize_particles() for particles.

### 3.11.1 Fortran Sample Code

The Fortran sample code given in the file sample.F90 is composed in a Fortran module named sample. It starts with the following lines to #include the header file ohhelp.f.h for level-3 function aliasing and to use the Fortran module ohhelp3 defined in oh_mod3.F90 for the interface's of level-3 and lower level library functions.

```fortran
#define OH_LIB_LEVEL 3
#include "ohhelp_f.h"
module sample
  use ohhelp3

Declaration

At first, we declare a few parameter's, MAXFRAC = 20 for maxfrac argument of oh3_init(), field-array identifiers for electromagnetic field-array eb(:,,:,:,:) (FEB = 1) and current density cd(:,,:,:,:) (FCD = 2), and element numbers of these arrays, EX, BX, JX and so on.

```fortran
implicit none
integer,parameter :: MAXFRAC=20
integer,parameter :: FEB=1,FCD=2
integer,parameter :: EX=1,EF=2,EZ=3,BX=4,BY=5,BZ=6
integer,parameter :: JX=1,JY=2,JZ=3
```
Then the variables to pass `oh3_init()` are declared with the same names as defined in Section 3.6.1. We also declare two field-arrays, `eb(:,:,,:,:)` for electromagnetic field and `cd(:,:,,:,:)` for current density.

```plaintext
type(oh_particle), allocatable :: pbuf(:)
integer :: sdid(2)
integer, allocatable :: nphgram(:,:,)
integer, allocatable :: totalp(:,:,)
type(oh_mycomm) :: mycomm
integer, allocatable :: sdoms(:,:,)
integer, allocatable :: bounds(:,:,)
type(oh_mycomm) :: mycomm
integer, allocatable :: sdoms(:,:,)
integer, allocatable :: bounds(:,:,)
type(oh_mycomm) :: mycomm
integer, allocatable :: sdoms(:,:,)
integer, allocatable :: bounds(:,:,)
type(oh_mycomm) :: mycomm
integer, allocatable :: sdoms(:,:,)
integer, allocatable :: bounds(:,:,)
```

The last declarative work is to give the prototypes of out-of-scope subroutines.

```plaintext
interface
subroutine initialize_eb(eb, sdom)
  implicit none
  real*8 :: eb(:,:,,:)
  integer :: sdom(:,:)
end subroutine
subroutine initialize_particles(pbuf, nspec, nphgram)
  use oh_type
  implicit none
  type(oh_particle) :: pbuf(:)
  integer :: nspec
  integer :: nphgram(:,:)
end subroutine
subroutine lorentz(eb, x, y, z, s, acc)
  implicit none
  real*8 :: eb(:,:,,:)
  real*8 :: x, y, z
  integer :: s
  real*8 :: acc(OH_DIMENSION)
end subroutine
subroutine scatter(p, s, c)
  use oh_type
  implicit none
  type(oh_particle) :: p
  integer :: s
  real*8 :: c(3,2,2,2)
end subroutine
subroutine rotate_b(eb, x, y, z, rot)
  implicit none
```

69
Subroutine pic()

The first subroutine pic() is the core of the simulator and is called with a few simulation parameters to be given to the arguments of oh3_init(), which are nspec, pcoord(3) and scoord(2,3). It also has arguments npmax for the absolute maximum number of the particle in the whole simulation and nstep to determine the number of simulation steps.

contains
  subroutine pic(nspec, pcoord, scoord, npmax, nstep)
  implicit none
  integer :: nspec
  integer :: pcoord(OH_DIMENSION)
  integer :: scoord(2, OH_DIMENSION)
  integer*8 :: npmax
  integer :: nstep

  integer :: n, t, maxlocalp, currmode

The first job is to allocate the array totalp(nspec,2) and a few other arrays having N as the size of a dimension, i.e., nphgram, sdoms and bounds. The number of nodes N = \( H_x \times H_y \times H_z \) is calculated from pcoord. We also allocate the particle array pbuf whose size maxlocalp is determined by oh2_max_local_particles() from npmax and MAXFRAC without additional minimum margin.

allocate(totalp(nspec,2))
  n = pcoord(1) * pcoord(2) * pcoord(3)
allocate(nphgram(n, nspec, 2))
allocate(sdoms(2, OH_DIMENSION, n))
allocate(bounds(2, OH_DIMENSION, n))

maxlocalp = oh_max_local_particles(nmax, MAXFRAC, 0)
allocation(pbuf(maxlocalp))

We continue initial setting of variables for oh3_init(); nbor and sdoms have the special values to delegate their initializations to oh3_init(); bcond indicates fully periodic boundary conditions by having 1s in all of its elements; the first element of ftypes for eb shows that the range for its broadcast is from \( eb(1,-1,-1,:) \) to \( eb(6,\sigma_x,\sigma_y,\sigma_z,:) \), while the second element for cd gives that for the reduction being from \( cd(1,-1,-1,:) \) to \( cd(3,\sigma_x+1,\sigma_y+1,\sigma_z+1,:) \); cfields has just two elements for eb and cd and thus their
communication type identifiers are same as their field identifiers; the first and second elements of ctypes for eb and cd are set as shown in Figure 13 and 14 respectively.

Now we can call oh3_init() and do it to have the sizes of field-arrays through ftypes by which we allocate the arrays eb and cd.

\[
\begin{align*}
nbor(1,1,1) &= -1 \\
sdoms(1,1,1) &= 0; \quad sdoms(2,1,1) = -1 \\
bcond(:, :) &= \text{reshape}((/1,1, 1, 1, 1/), (/2, OH\_DIMENSION/)) \\
types(:, FEB) &= (/6, 0, 0, -1, 1, 0,0/) \quad ! \text{for eb}() \\
types(:, FCD) &= (/3, 0, 0, 0, 0, -1,2/) \quad ! \text{for cd}() \\
types(1, FCD+1) &= -1 \quad ! \text{terminator} \\
cfields(:, :) &= (/FEB,FCD,0/) \\
cypes(:, 1, FEB) &= \text{reshape}((/ 0,0,2, -1,-1,1/), (/3,2/)) \quad ! \text{for eb}() \\
cypes(:, 1, FCD) &= \text{reshape}((/-1,2,3, -1,-4,3/), (/3,2/)) \quad ! \text{for cd}() \\
call oh\_init(sdid(:,), nspec, MAXFRAC, nphgram(:,,:), totalp(:,,:), \& pbuf(:,), phase(:,), maxlocalp, mycomm, nbor(:,,:), \& pcoord(:,), sdoms(:,,:), scoord(:,,:), 1, bcond(:,,:), \& bounds(:,,:), ftypes(:,,:), cfields(:,), ctypes(:,,:), \& fsizes(:,,:), 0, 0, 0) \\
allocate(eb(6, fsizes(1,1,FEB):fsizes(2,1,FEB), \& fsizes(1,2,FEB):fsizes(2,2,FEB), \& fsizes(1,3,FEB):fsizes(2,3,FEB, 2)) \\
allocate(cd(3, fsizes(1,1,FCD):fsizes(2,1,FCD), \& fsizes(1,2,FCD):fsizes(2,2,FCD), \& fsizes(1,3,FCD):fsizes(2,3,FCD, 2))
\end{align*}
\]

We still have a few initializations to have initial setting of eb for primary subdomain, whose size and location in the space domain is given in sdoms(:,sdid(1)), by the out-of-scope subroutine initialize_eb(), and that of primary particles in pbuf and the count for each of nspec species and each of subdomain in nphgram(:,1) by the out-of-scope subroutine initialize_particles()\textsuperscript{19}. Then we call oh3_transbound() to examine whether the initial particle positioning is balanced and, if not, broadcast eb to the helpers of the local node by oh3_bcast_field()\textsuperscript{20}. Finally, the boundary values of initial setting of eb are exchanged between adjacent nodes by oh3_exchange_borders().

\[
\begin{align*}
call initialize\_eb(eb(:,,:,1), sdoms(:,sdid(1))) \\
call initialize\_particles(pbuf(:,), nspec, nphgram(:,1)) \\
currmode = oh\_transbound(0, 0) \\
\text{if (currmode.lt.0) then} \\
\quad call oh\_bcast\_field(eb(1,0,0,0,1), eb(1,0,0,0,2), FEB) \\
\quad currmode = 1 \\
\text{end if} \\
call oh\_exchange\_borders(eb(1,0,0,0,1), eb(1,0,0,0,2), FEB, currmode)
\end{align*}
\]

\textsuperscript{19}It might need other parameters to initialize pbuf, e.g., the number of initial particles of each species as a whole, but such parameters are also out-of-scope.

\textsuperscript{20}Broadcasting from the local subdomain coordinates \((-1, -1, -1)\) to \((\sigma_x, \sigma_y, \sigma_z)\) is a little bit larger than what we really need because oh3_exchange_borders just follows, but it is safe and the additional communication cost is negligible.
Now we start the main loop of simulation. First, we call \texttt{particle\_push()} giving it primary particles and the electromagnetic field-array \texttt{eb} of primary subdomain. Then, if the local node has secondary particles and subdomain, i.e., \texttt{sdid(2)} for its secondary subdomain identifier is not negative, we call the subroutine again giving it secondary particles and the field-array of secondary subdomain. Then we call \texttt{oh3\_transbound()} to transfer particles among nodes and, if it (re)built the helpand-helper configuration, \texttt{oh3\_bcast\_field()} to broadcast \texttt{eb} to helpers.

\begin{verbatim}
do t=1, nstep
   call particle\_push(pbuf(pbase(1):), nspec, totalp(:,1), &
                      eb(:,:,1,:), sdoms(:,:,sdid(1)), sdid(1), 0, &
                      nphgram(:,:,1))
   if (sdid(2).ge.0) &
      call particle\_push(pbuf(pbase(2):), nspec, totalp(:,2), &
                         eb(:,:,2,:), sdoms(:,:,sdid(2)), sdid(2), 1, &
                         nphgram(:,:,2))
   currmode = oh\_transbound(currmode, 0)
   if (currmode.lt.0) then
      call oh\_bcast\_field(eb(1,0,0,0,1), eb(1,0,0,0,2), FEB)
      currmode = 1
   end if

Next we call \texttt{current\_scatter()} once or twice giving it primary and secondary particles and the field-array \texttt{cd} of subdomains, to have current density vectors in the primary subdomain, or a partial results of them in primary and secondary subdomains if we are in secondary mode. In the latter case, we call \texttt{oh3\_allreduce\_field()} to have almost complete sums of the vectors in both primary and secondary subdomains. Then, to obtain the contribution of the particles near by the subdomain boundaries and residing (or having resided) in adjacent subdomains, we call \texttt{oh3\_exchange\_borders()} to have the boundary values of \texttt{cd}, and \texttt{add\_boundary\_current()} to add them to those calculated by the local node. If the local node has the secondary subdomain, \texttt{add\_boundary\_current()} is called twice, one for the primary subdomain and the other for the secondary.

\begin{verbatim}
call current\_scatter(pbuf(pbase(1):), nspec, totalp(:,1), &
                    cd(:,:,1,:), sdoms(:,:,sdid(1)), &
                    ctypes(:,:,1,FCD))
   if (sdid(2).ge.0) &
      call current\_scatter(pbuf(pbase(2):), nspec, totalp(:,2), &
                         cd(:,:,2,:), sdoms(:,:,sdid(2)), &
                         ctypes(:,:,1,FCD))
   if (currmode.ne.0) &
      call oh\_allreduce\_field(cd(1,0,0,0,1), cd(1,0,0,0,2), FCD)
      call oh\_exchange\_borders(cd(1,0,0,0,1), cd(1,0,0,0,2), FCD, currmode)
      call add\_boundary\_current(cd(:,:,1,:), sdoms(:,:,sdid(1)), &
                                ctypes(:,:,1,FCD))
   if (sdid(2).ge.0) &
      call add\_boundary\_current(cd(:,:,2,:), sdoms(:,:,sdid(2)), &
                                ctypes(:,:,1,FCD))
\end{verbatim}

Next, we update field vectors \(E\) and \(B\) in the primary subdomain by calling \texttt{field\_solve\_e()} and \texttt{field\_solve\_b()} respectively, giving them the field-arrays of the primary
subdomain. Then, if the local node has the secondary subdomain, we call these two
routines again giving them field-arrays of the secondary subdomain. Finally, the boundary
values of \( eb \) are exchanged between adjacent subdomains by \texttt{oh3\_exchange\_borders()} to
have what we need in the next simulation step.

\begin{verbatim}
call field\_solve\_e(eb(:,:,,:,1), cd(:,:,,:,1), sdoms(:,:,sdid(1)))
call field\_solve\_b(eb(:,:,,:,1), sdoms(:,:,sdid(1)))
if (sdid(2).ge.0) then
  call field\_solve\_e(eb(:,:,,:,2), cd(:,:,,:,2), sdoms(:,:,sdid(2)))
call field\_solve\_b(eb(:,:,,:,2), sdoms(:,:,sdid(2)))
end if
end subroutine
\end{verbatim}

\textbf{Subroutine \texttt{particle\_push}()} 

The second subroutine \texttt{particle\_push()} is given eight arguments to specify primary or
secondary particles, primary or secondary subdomain and its field-array; \texttt{pbuf} for particle
buffer; \texttt{nsp} for the number of species; \texttt{totalp} for the number of particles in each species;
\texttt{eb} for the electromagnetic field-array; \texttt{sdom} for the size and the location of the subdomain;
\texttt{n} for the subdomain identifier; \texttt{ps} for primary or secondary mode; and \texttt{nphgram} for the
particle population histogram.

\begin{verbatim}
subroutine particle\_push(pbuf, nspec, totalp, eb, sdom, n, ps, nphgram)
implicit none
type(oh\_particle) :: pbuf(:)
integer :: nspec
integer :: totalp(:)
real*8 :: eb(:,:,:,:)
integer :: sdom(:,:)
integer :: n
integer :: ps
integer :: nphgram(:,:)
integer :: xl, yl, zl, xu, yu, zu
integer :: s, p, q, m
real*8 :: acc(OH\_DIMENSION)

Before we enter the double loop for species and particles in each of them, we get lower
and upper subdomain boundaries from \texttt{sdom} to set them into \texttt{xl}, \texttt{xu} and so on, for the sake
of conciseness (and efficiency if your compiler is not smart enough).

\texttt{xl=sdom(1,1); yl=sdom(1,2); zl=sdom(1,3)}
\texttt{xu=sdom(2,1); yu=sdom(2,2); zu=sdom(2,3)}

Now we start the double loop letting \texttt{nphgram(n+1,s)} have \texttt{totalp(s)} as its initial value
at the beginning of the iteration for each species \texttt{s}, to mean that we will have \texttt{totalp(s)}
in the subdomain \texttt{n} if all the particles of the species \texttt{s} stay in the subdomain. Then we
call \texttt{lorentz()} to have the acceleration vector of each particle in the array \texttt{acc(3)}, whose
elements are added to the velocity vector components of the particle. After this acceleration
(or deceleration), the particle is moved by adding the velocity vector to the position vector.

73


```
p = 0
  do s = 1, nspec
    nphgram(n+1,s) = totalp(s)
    do q = 1, totalp(s)
      p = p + 1
      call lorentz(eb, pbuf(p)%x-xl, pbuf(p)%y-yl, pbuf(p)%z-zl, s, acc)
      pbuf(p)%vx = pbuf(p)%vx + acc(1)
      pbuf(p)%vy = pbuf(p)%vy + acc(2)
      pbuf(p)%vz = pbuf(p)%vz + acc(3)
      pbuf(p)%x = pbuf(p)%x + pbuf(p)%vx
      pbuf(p)%y = pbuf(p)%y + pbuf(p)%vy
      pbuf(p)%z = pbuf(p)%z + pbuf(p)%vz
    end do
  end do
end subroutine
```

Now we finish the job for a particle if it is still staying in the subdomain. Otherwise, we call `oh3_map_particle_to_neighbor()` to obtain the identifier `m` of the subdomain in which the particle now resides. Then `nphgram(n+1,s)` is decreased by one to indicate that the particle has gone, while `nphgram(m+1,s)` is increased by one to represent its immigration. We also update `nid` element of the particle to show it now resides in the subdomain `m`.

```
if (pbuf(p)%x.lt.xl .or. pbuf(p)%x.ge.xu .or. &
    pbuf(p)%y.lt.yl .or. pbuf(p)%y.ge.yu .or. &
    pbuf(p)%z.lt.zl .or. pbuf(p)%z.ge.zu) then
  m = oh_map_particle_to_neighbor(pbuf(p)%x, pbuf(p)%y, pbuf(p)%z, ps)
  nphgram(n+1,s) = nphgram(n+1,s) - 1
  nphgram(m+1,s) = nphgram(m+1,s) + 1
  pbuf(p)%nid = m
end if
end do
end subroutine
```

**Subroutine `current_scatter()`**

The third subroutine `current_scatter()` is given six arguments to specify primary or secondary particles, primary or secondary subdomain and its field-array; `pbuf` for particle buffer; `nspec` for the number of species; `totalp` for the number of particles in each species; `cd` for the field-array of current density vectors; `sdom` for the size and the location of the subdomain; and `ctype` to know the range in `cd` which the particles will contribute to.

```
subroutine current_scatter(pbuf, nspec, totalp, cd, sdom, ctype)
  implicit none
  type(oh_particle) :: pbuf(:)
  integer :: nspec
  integer :: totalp(:)
  real*8 :: cd(:,:,:,:)
  integer :: sdom(:,:)
  integer :: ctype(3,2)
  integer :: xl, yl, zl, xu, yu, zu
  integer :: s, p, q
  integer :: i, j, k
  real*8 :: x, y, z
```

74
Before we enter the double loop for species and particles in each of them, we get lower subdomain boundaries from \texttt{sdom} to set them into \texttt{xl} and so on, and upper boundaries to set those in the local subdomain coordinates into \texttt{xu} and so on, for the sake of conciseness. Then we zero-clear \texttt{cd} including the boundary planes we will send to adjacent nodes referring to \texttt{ctype}.

\begin{verbatim}
xl = sdom(1,1); yl = sdom(1,2); zl = sdom(1,3)
xu = sdom(2,1)-xl; yu = sdom(2,2)-yl; zu = sdom(2,3)-zl
do k=ctype(1,1), zu+ctype(1,2)+ctype(1,3)-1
do j=ctype(1,1), yu+ctype(1,2)+ctype(1,3)-1
do i=ctype(1,1), xu+ctype(1,2)+ctype(1,3)-1
  cd(JX, i, j, k) = 0.0d0
  cd(JY, i, j, k) = 0.0d0
  cd(JZ, i, j, k) = 0.0d0
end do; end do; end do
\end{verbatim}

Now we start the double loop. In each iteration for a particle, we call \texttt{scatter()} to have its contribution to the current density vectors of the grid points surrounding it in the array \texttt{c(3,2,2,2)}, whose elements are added to the corresponding elements of \texttt{cd}.

\begin{verbatim}
p = 0
do s=1, nspec
  do q=1, totalp(s)
    p = p + 1
    call scatter(pbuf(p), s, c)
    x = pbuf(p)%x - xl; y = pbuf(p)%y - yl; z = pbuf(p)%z - zl
    do k=0,1; do j=0,1; do i=0,1
      cd(JX, x+i, y+j, z+k) = cd(JX, x+i, y+j, z+k) + c(JX, i, j, k)
      cd(JY, x+i, y+j, z+k) = cd(JY, x+i, y+j, z+k) + c(JY, i, j, k)
      cd(JZ, x+i, y+j, z+k) = cd(JZ, x+i, y+j, z+k) + c(JZ, i, j, k)
    end do; end do; end do;
  end do
end do
end subroutine
\end{verbatim}

**Subroutine \texttt{add_boundary_current}()**

The fourth subroutine \texttt{add_boundary_current()} is given three arguments to specify the primary or secondary subdomain and its field-array; \texttt{cd} for the field-array of current density vectors; \texttt{sdom} for the size and the location of the subdomain; and \texttt{ctype} to know the boundary planes in \texttt{cd}.

\begin{verbatim}
subroutine add_boundary_current(cd, sdom, ctype)
  implicit none
  real*8 :: cd(:,:,:,:)
  integer :: sdom(2,OH_DIMENSION)
  integer :: ctype(3,2)
  integer :: xu, yu, zu
  integer :: sl, dl, nl, su, du, nu
\end{verbatim}
First, we calculate the upper boundaries \( \sigma_{x,y,z} \) of the subdomain in its local coordinates referring to \( s_{dom} \) and set them into \( x_u \) and so on. Then, to calculate the base (lowest coordinate) of the boundary planes, \( s^l_{x,y,z} \) and \( s^u_{x,y,z} \) for the planes obtained from neighbors and \( d^l_{x,y,z} \) and \( d^u_{x,y,z} \) for those to add to, and the number of lower and upper boundary planes \( n_l \) and \( n_u \), we refer to \( \text{ctype} \) elements to have the followings:

\[
\begin{align*}
  s^l_{x,y,z} &= \text{ctype}(2,2) \\
  s^u_{x,y,z} &= \sigma_{x,y,z} + \text{ctype}(2,1) \\
  n_l &= \text{ctype}(3,2) \\
  d^l_{x,y,z} &= s^l_{x,y,z} + n_l \\
  n_u &= \text{ctype}(3,1) \\
  d^u_{x,y,z} &= s^u_{x,y,z} - n_u
\end{align*}
\]

That is, we suppose the planes to add to are at just \( \text{inside} \) of the planes obtained from neighbors.

\[
\begin{align*}
  x_u &= s_{dom}(2,1) - s_{dom}(1,1) \\
  y_u &= s_{dom}(2,2) - s_{dom}(1,2) \\
  z_u &= s_{dom}(2,3) - s_{dom}(1,3) \\
  s_l &= \text{ctype}(2,2); \ n_l = \text{ctype}(3,2); \ d_l = s_l + n_l \\
  s_u &= \text{ctype}(2,1); \ n_u = \text{ctype}(3,1); \ d_u = s_u - n_u
\end{align*}
\]

Then we call \( \text{add\_boundary\_curr()} \) six times for lower and upper boundary planes perpendicular to \( z, y \) and \( x \) axes in this order to do the followings conceptually:

\[
\begin{align*}
  [s^l_x, s^u_x + n_u] \times [s^l_y, s^u_y + n_u] \times (d^l_z, d^u_z + n_l) & \leftarrow \\
  [s^l_x, s^u_x + n_u] \times [s^l_y, s^u_y + n_u] \times (d^l_z, d^u_z + n_l) \leftarrow \\
  [s^l_x, s^u_x + n_u] \times [s^l_y, s^u_y + n_u] \times (d^l_z, d^u_z + n_l) \leftarrow \\
  [s^l_x, s^u_x + n_u] \times [s^l_y, s^u_y + n_u] \times (d^l_z, d^u_z + n_l) \leftarrow \\
  [s^l_x, s^u_x + n_u] \times [s^l_y, s^u_y + n_u] \times (d^l_z, d^u_z + n_l) \leftarrow \\
  [s^l_x, s^u_x + n_u] \times [s^l_y, s^u_y + n_u] \times (d^l_z, d^u_z + n_l) \leftarrow \\
  [d^l_x, d^u_x + n_l] \times [d^l_y, d^u_y + n_u] \times (d^l_z, d^u_z + n_u) \leftarrow \\
  [d^l_x, d^u_x + n_l] \times [d^l_y, d^u_y + n_u] \times (d^l_z, d^u_z + n_u) \leftarrow \\
  [d^l_x, d^u_x + n_l] \times [d^l_y, d^u_y + n_u] \times (d^l_z, d^u_z + n_u) \leftarrow \\
  [d^l_x, d^u_x + n_l] \times [d^l_y, d^u_y + n_u] \times (d^l_z, d^u_z + n_u) \leftarrow \\
  [d^l_x, d^u_x + n_l] \times [d^l_y, d^u_y + n_u] \times (d^l_z, d^u_z + n_u) \leftarrow \\
\end{align*}
\]

The operations above for a two-dimensional subdomain are illustrated in Figure 15.
Figure 15: Adding boundary planes of current density vectors.

call add_boundary_curr(sl, dl, nl, &
dl, dl, yu+(du-dl), &
dl, dl, zu+(du-dl), cd)
call add_boundary_curr(xu+su, xu+du, nu, &
dl, dl, yu+(du-dl), &
dl, dl, zu+(du-dl), cd)
end subroutine

Subroutine add_boundary_curr()
The fifth subroutine add_boundary_curr() does the followings conceptually for each current density vector component in cd for the boundary plane addition in add_boundary_current().

\[
[x_d, x_d + n_x] \times [y_d, y_d + n_y] \times [z_d, z_d + n_z] \leftarrow [x_d, x_d + n_x] \times [y_d, y_d + n_y] \times [z_d, z_d + n_z] + [x_s, x_s + n_x] \times [y_s, y_s + n_y] \times [z_s, z_s + n_z]
\]

subroutine add_boundary_curr(xs, xd, nx, ys, yd, ny, zs, zd, nz, cd)
    implicit none
    integer :: xs, xd, nx, ys, yd, ny, zs, zd, nz
    integer :: i, j, k
    real*8 :: cd(:,:,:,:)
    do k=0, nz-1; do j=0, ny-1; do i=0, nx-1
        cd(JX, xd+i, yd+j, zd+k) = &
            cd(JX, xd+i, yd+j, zd+k) + cd(JX, xs+i, ys+j, zs+k)
        cd(JY, xd+i, yd+j, zd+k) = &
            cd(JY, xd+i, yd+j, zd+k) + cd(JY, xs+i, ys+j, zs+k)
        cd(JZ, xd+i, yd+j, zd+k) = &
            cd(JZ, xd+i, yd+j, zd+k) + cd(JZ, xs+i, ys+j, zs+k)
    end do; end do; end do
end subroutine
Subroutine field\_solve\_e()

The sixth subroutine field\_solve\_e() is given three arguments to specify the primary or secondary subdomain and its field-arrays; eb for the electromagnetic field-array; cd for the field-array of current density vectors; and sdom for the size and the location of the subdomain.

```fortran
subroutine field\_solve\_e(eb, cd, sdom)
  implicit none
  real*8 :: eb(:,,:,:)
  real*8 :: cd(:,,:,:)
  integer :: sdom(2,OH\_DIMENSION)
  integer :: xu, yu, zu, x, y, z
  real*8 :: rot(OH\_DIMENSION)

  First, we calculate the upper boundaries $\sigma_{x,y,z}$ of the subdomain in its local coordinates referring to sdom and set them into xu and so on. Then, in the loop for $[0, \sigma_x] \times [0, \sigma_y] \times [0, \sigma_z]$, we update each electric field vector following the Maxwell’s (or Ampère’s circuital) law using $\nabla \times B$ calculated by the out-of-scope subroutine rotate\_b() and set into rot(3), and the current density vectors cd. Note that the constants EPSILON for $\varepsilon_0$ and MU for $\mu_0$ are assumed to have been defined somewhere in the simulation code.

  xu = sdom(2,1) - sdom(1,1)
  yu = sdom(2,2) - sdom(1,2)
  zu = sdom(2,3) - sdom(1,3)
  do z=0, zu; do y=0, yu; do x=0, xu
    call rotate\_b(eb(:,,:,:,:), x, y, z, rot)
    eb(EX, x, y, z) = eb(EX, x, y, z) + &
      (1/EPSILON)*((1/MU)*rot(1) + cd(JX, x, y, z))
    eb(EY, x, y, z) = eb(EY, x, y, z) + &
      (1/EPSILON)*((1/MU)*rot(2) + cd(JY, x, y, z))
    eb(EZ, x, y, z) = eb(EZ, x, y, z) + &
      (1/EPSILON)*((1/MU)*rot(3) + cd(JZ, x, y, z))
  end do; end do; end do
end subroutine
```

Subroutine field\_solve\_b()

The seventh and last subroutine field\_solve\_b() is given two arguments to specify the primary or secondary subdomain and its field-array; eb for the electromagnetic field-array; and sdom for the size and the location of the subdomain.

```fortran
subroutine field\_solve\_b(eb, sdom)
  implicit none
  real*8 :: eb(:,,:,:)
  integer :: sdom(2,OH\_DIMENSION)
  integer :: xu, yu, zu, x, y, z
  real*8 :: rot(OH\_DIMENSION)

  First, we calculate the upper boundaries $\sigma_{x,y,z}$ of the subdomain in its local coordinates referring to sdom and set them into xu and so on. Then, in the loop for $[0, \sigma_x-1] \times \ldots \times [0, \sigma_z]$, we update each electric field vector following the Maxwell’s law using $\nabla \times B$ calculated by the out-of-scope subroutine rotate\_b() and set into rot(3), and the current density vectors cd. Note that the constants EPSILON for $\varepsilon_0$ and MU for $\mu_0$ are assumed to have been defined somewhere in the simulation code.

  xu = sdom(2,1) - sdom(1,1)
  yu = sdom(2,2) - sdom(1,2)
  zu = sdom(2,3) - sdom(1,3)
  do z=0, zu; do y=0, yu; do x=0, xu
    call rotate\_b(eb(:,,:,:,:), x, y, z, rot)
    eb(EX, x, y, z) = eb(EX, x, y, z) + &
      (1/EPSILON)*((1/MU)*rot(1) + cd(JX, x, y, z))
    eb(EY, x, y, z) = eb(EY, x, y, z) + &
      (1/EPSILON)*((1/MU)*rot(2) + cd(JY, x, y, z))
    eb(EZ, x, y, z) = eb(EZ, x, y, z) + &
      (1/EPSILON)*((1/MU)*rot(3) + cd(JZ, x, y, z))
  end do; end do; end do
end subroutine
```
we update each magnetic field vector following the Maxwell’s (or Faraday’s induction) law using $\nabla \times \mathbf{E}$ calculated by the out-of-scope subroutine \texttt{rotate\_e()} and set into \texttt{rot(3)}.

\begin{verbatim}
xu = sdom(2,1) - sdom(1,1)  
yu = sdom(2,2) - sdom(1,2)  
zu = sdom(2,3) - sdom(1,3)  
do z=0, zu-1; do y=0, yu-1; do x=0, xu-1  
call rotate\_e(eb(:, :, :, :), x, y, z, rot)  
eb(BX, x, y, z) = eb(BX, x, y, z) + rot(1)  
eb(BY, x, y, z) = eb(BY, x, y, z) + rot(2)  
eb(BZ, x, y, z) = eb(BZ, x, y, z) + rot(3)  
end do; end do; end do
\end{verbatim}

end subroutine

end module

3.11.2 C Sample Code

The C sample code is given in the file sample.c. It starts with the following lines to \#include the header file \texttt{ohhelp\_c\_h} for level-3 function aliasing and prototypes of level-3 and lower level library functions. It also \#include’s the standard header file \texttt{stdlib.h} for \texttt{malloc()}.

\begin{verbatim}
#include <stdlib.h>
#define OH_LIB_LEVEL 3
#include "ohhelp\_c\_h"
\end{verbatim}

Declaration

At first, we \#define a few constants, \texttt{MAXFRAC} = 20 for \texttt{maxfrac} argument of \texttt{oh3\_init()}, field-array identifiers for electromagnetic field-array \texttt{eb[]} (FEB = 0) and current density \texttt{cd[]} (FCD = 1).

\begin{verbatim}
#define MAXFRAC 20
#define FEB 0
#define FCD 1
\end{verbatim}

Then the variables to pass \texttt{oh3\_init()} are declared with the same names as defined in Section 3.6.1 and a part of them are initialized as follows; pointers \texttt{pbuf}, \texttt{nbor}, \texttt{sdoms} and \texttt{bounds} have NULL to make \texttt{oh3\_init()} allocate them and initialize the last three in the default manner; \texttt{bcond} indicates fully periodic boundary conditions by having 0s in all of its elements; the first element of \texttt{ftypes} for \texttt{eb} shows that the range for its broadcast is from the local subdomain coordinates $(-1, -1, -1)$ to $(\sigma_x, \sigma_y, \sigma_z)$ while the second element for \texttt{cd} gives that for the reduction being from $(-1, -1, -1)$ to $(\sigma_x+1, \sigma_y+1, \sigma_z+1)$; \texttt{cfields} has just two elements for \texttt{eb} and \texttt{cd} and thus their communication type identifiers are same as their field identifiers; the first and second elements of \texttt{ctypes} for \texttt{eb} and \texttt{cd} are set as shown in Figure 13 and 14 respectively. We also declare two pointer arrays to field-arrays, \texttt{eb[2]} for electromagnetic field and \texttt{cd[2]} for current density, together with their \texttt{struct} namely \texttt{ebfield} and \texttt{current}.

\begin{verbatim}
int sdid[2];
int **nphgram[2];
\end{verbatim}
Another declarative work is to give the prototypes of functions defined in this source file and of out-of-scope ones.

/* prototypes of functions defined in sample.c */
void pic(int nspec, int pcoord[OH_DIMENSION], int scoord[OH_DIMENSION][2],
         long long int npmax, int nstep);
void particle_push(struct S_particle *pbuf, int nspec, int *totalp,
                   struct ebfield *eb, int sdom[OH_DIMENSION][2],
                   int fsize[OH_DIMENSION][2], int n, int ps, int **nphgram);
void current_scatter(struct S_particle *pbuf, int nspec, int *totalp,
                     struct current *cd, int sdom[OH_DIMENSION][2],
                     int ctype[2][3], int fsize[OH_DIMENSION][2]);
void add_boundary_current(struct current *cd, int sdom[OH_DIMENSION][2],
                          int ctype[2][3], int fsize[3][2]);
void field_solve_e(struct ebfield *eb, struct current *cd,
                    int sdom[OH_DIMENSION][2], int fsizee[OH_DIMENSION][2],
                    int fsizec[OH_DIMENSION][2]);
void field_solve_b(struct ebfield *eb, int sdom[OH_DIMENSION][2],
                   int fsizeb[OH_DIMENSION][2]);

/* prototypes of functions not defined in sample.c */
void initialize_efield(struct ebfield *eb, int sdom[OH_DIMENSION][2],
                       int fsizee[OH_DIMENSION][2]);
void initialize_particles(struct S_particle *pbuf, int nspec, int **nphgram);
void lorentz(struct ebfield *eb, double x, double y, double z, int s,
             int fsizeb[OH_DIMENSION][2], double acc[OH_DIMENSION]);
void scatter(struct S_particle p, int s, struct current c[2][2][2]);
void rotate_b(struct ebfield *eb, double x, double y, double z, int fsize[OH_DIMENSION][2], double rot[OH_DIMENSION]);
void rotate_e(struct ebfield *eb, double x, double y, double z, int fsize[OH_DIMENSION][2], double rot[OH_DIMENSION]);

The last declarative work is to define two functional macros field_array_size(FS) and malloc_field_array(S,FS). The former is to calculate the number of elements in an array of conceptually three dimensional but one-dimensional in reality from FS being a subarray of fsizes[] [OH_DIMENSION] [2] reported from oh3_init(). The latter is to malloc() a field-array whose element is a struct named S and whose size is given by FS being a subarray of fsizes. These macros are for a concise implementation of what we described in Section 3.6.1.

#define field_array_size(FS) 
((FS[0][1]-FS[0][0])*(FS[1][1]-FS[1][0])*(FS[2][1]-FS[2][0]))
#define malloc_field_array(S,FS) 
((struct S*)malloc(sizeof(struct S)*field_array_size(FS)*2)- 
FS[0][0]+(FS[0][1]-FS[0][0])*(FS[1][0]+(FS[1][1]-FS[1][0])*FS[2][0]))

Function pic()

The first function pic() is the core of the simulator and is called with a few simulation parameters to be given to the arguments of oh3_init(), which are nspec, pcoord[3] and scoord[3] [2]. The function also has arguments npmax for the absolute maximum number of the particles in the whole simulation and nstep to determine the number of simulation steps.

void pic(int nspec, int pcoord[OH_DIMENSION], int scoord[OH_DIMENSION][2], long long int npmax, int nstep) {
int n, i, j, t;
int currmode;

The first job is the allocation of the bodies of totalp and nphgram, which we could depute oh3_init() to do but in this example we dare to do for the sake of clarity. The allocation for the former is fairly simple because we just need an one-dimensional array of S × 2 and make totalp[0] and totalp[1] point its element [0] and [S]. The allocation for the later is a little bit more complicated as exemplified in Section 3.2.4. Its size N for the number of nodes N = H_x × H_y × H_z is calculated from pcoord.

totalp[0] = (int*)malloc(sizeof(int)*nspec*2);
totalp[1] = totalp[0] + npmax;
n = pcoord[0] * pcoord[1] * pcoord[2];
nphgram[0] = (int**)malloc(sizeof(int*)*nspec*2);
nphgram[1] = nphgram[0] + npmax;
nphgram[0][0] = (int*)malloc(sizeof(int)*n*nspec*2);
nphgram[1][0] = nphgram[0][0] + n*nspec;
for (i=0; i<n; i++) for (j=1; j<nspec; j++)
    nphgram[i][j] = nphgram[i][j-1] + n;

81
Now we can call oh3_init() and do it giving the size of pbuf calculated by oh2_max_local_particles() to its argument maxlocalp, and NULL to mycomm because it is unnecessary. Then, with the sizes of field-arrays given through ftypes, we allocate the arrays so that they are pointed by eb and cd using the macros malloc_field_array() and field_array_size().

```c
oh_init((int**)(&sdid), nspec, MAXFRAC, mphgram[0], totalp, &pbuf,
    (int**)(&pbase), oh_max_local_particles(nmax, MAXFRAC, 0), NULL,
    &nbor, pcoord, (int**)(&sdoms), &scoord[0][0], 1, &bcond[0][0],
    &bounds, ftypes[0], cfields, ctypes[0][0][0], (int**)(&fsizes),
    0, 0, 0);
```

```c
eb[0] = malloc_field_array(ebfield, fsizes[FEB]);
eb[1] = eb[0] + field_array_size(fsizes[FEB]);
cd[0] = malloc_field_array(current, fsizes[FCD]);
cd[1] = cd[0] + field_array_size(fsizes[FCD]);
```

We still have a few initializations to have initial setting of eb for primary subdomain, whose size and location in the space domain is given in sdoms[sdid[0]][][] by the out-of-scope function initialize_eb(), and that of primary particles in pbuf and the count for each of nspec species and each of subdomain in mphgram[0][] by the out-of-scope function initialize_particles().

Note that initialize_eb() is also given fsizes[FEB][] as its argument to calculate one-dimensional indices of eb. Then we call oh3_transbound() to examine whether the initial particle positioning is balanced and, if not, broadcast eb to the helpers of the local node by oh3_bcast_field(). Finally, the boundary values of initial setting of eb are exchanged between adjacent nodes by oh3_exchange_borders().

```c
initialize_eb(eb[0], sdoms[sdid[0]], fsizes[FEB]);
initialize_particles(pbuf, nspec, mphgram[0]);
```

```c
currmode = oh_transbound(0, 0);
if (currmode<0) {
    oh_bcast_field(eb[0], eb[1], 0); currmode = 1;
}
oh_exchange_borders(eb[0], eb[1], FEB, currmode);
```

Now we start the main loop of simulation. First, we call particle_push() giving it primary particles and the electromagnetic field-array eb of primary subdomain. Then, if the local node has secondary particles and subdomain, i.e., sdid[1] for its secondary subdomain identifier is not negative, we call the function again giving it secondary particles and the field-array of secondary subdomain. Then we call oh3_transbound() to transfer particles among nodes and, if it (re)built the helpand-helper configuration, oh3_bcast_field() to broadcast eb to helpers.

```c
for (t=0; t<nstep; t++) {
```

21 It might need other parameters to initialize pbuf, e.g., the number of initial particles of each species as a whole, but such parameters are also out-of-scope.

22 Broadcasting from the local subdomain coordinates (−1, −1, −1) to (σx, σy, σz) is a little bit larger than what we really need because oh3_exchange_borders just follows, but it is safe and the additional communication cost is negligible.
particle_push(pbuf+pbase[0], nspec, totalp[0], eb[0], sdoms[sdid[0]],
fsizes[FEB], sdid[0], 0, nphgram[0]);
if (sdid[1]>=0)
particle_push(pbuf+pbase[1], nspec, totalp[1], eb[1], sdoms[sdid[1]],
fsizes[FEB], sdid[1], 1, nphgram[1]);
currmode = oh_transbound(0, 0);
if (currmode<0) {
    oh_bcast_field(eb[0], eb[1], 0); currmode = 1;
}

Next we call current_scatter() once or twice giving it primary and secondary particles and the field-array cd of subdomains, to have current density vectors in the primary subdomain, or a partial results of them in primary and secondary subdomains if we are in secondary mode. In the latter case, we call oh3_allreduce_field() to have almost complete sums of the vectors in both primary and secondary subdomains. Then, to obtain the contribution of the particles near by the subdomain boundaries and residing (or having resided) in adjacent subdomains, we call oh3_exchange_borders() to have the boundary values of cd, and add_boundary_current() to add them to those calculated by the local node. If the local node has the secondary subdomain, add_boundary_current() is called twice, one for the primary subdomain and the other for the secondary.

current_scatter(pbuf+pbase[0], nspec, totalp[0], cd[0], sdoms[sdid[0]],
ctypes[FCD][0], fsizes[FCD]);
if (sdid[1]>=0)
current_scatter(pbuf+pbase[1], nspec, totalp[1], cd[1], sdoms[sdid[1]],
ctypes[FCD][0], fsizes[FCD]);
if (currmode) oh_allreduce_field(cd[0], cd[1], FCD);
oh_exchange_borders(cd[0], cd[1], FCD, currmode);
add_boundary_current(cd[0], sdoms[sdid[0]], ctypes[FCD][0], fsizes[FCD]);
if (sdid[1]>=0)
add_boundary_current(cd[1], sdoms[sdid[1]], ctypes[FCD][0], fsizes[FCD]);

Next, we update field vectors $E$ and $B$ in the primary subdomain by calling field_solve_e() and field_solve_b() respectively, giving them the field-arrays of the primary subdomain. Then, if the local node has the secondary subdomain, we call these two functions again giving them field-arrays of the secondary subdomain. Finally, the boundary values of eb are exchanged between adjacent subdomains by oh3_exchange_borders() to have what we need in the next simulation step.

field_solve_e(eb[0], cd[0], sdoms[sdid[0]], fsizes[FEB], fsizes[FCD]);
field_solve_b(eb[0], sdoms[sdid[0]], fsizes[FEB]);
if (sdid[1]>=0) {
    field_solve_e(eb[1], cd[1], sdoms[sdid[1]], fsizes[FEB], fsizes[FCD]);
    field_solve_b(eb[1], sdoms[sdid[1]], fsizes[FEB]);
}
    oh_exchange_borders(eb[0], eb[1], FEB, currmode);
}
Function particle_push()

The second function `particle_push()` is given nine arguments to specify primary or secondary particles, primary or secondary subdomain and its field-array; `pbuf` for particle buffer; `nspec` for the number of species; `totalp` for the number of particles in each species; `eb` for the electromagnetic field-array; `sdom` for the size and the location of the subdomain; `fsize` for the size of `eb`; `n` for the subdomain identifier; `ps` for primary or secondary mode; and `nphgram` for the particle population histogram.

Then, in the local variable declaration, we get lower and upper subdomain boundaries from `sdom` to set them into `xl`, `xu` and so on, for the sake of conciseness (and efficiency if your compiler is not smart enough).

```c
void particle_push(struct S_particle *pbuf, int nspec, int *totalp, 
                    struct ebfield *eb, int sdom[OH_DIMENSION][2], 
                    int fsize[OH_DIMENSION][2], int n, int ps, int **nphgram) {

    int xl=sdom[0][0], yl=sdom[1][0], zl=sdom[2][0];
    int xu=sdom[0][1], yu=sdom[1][1], zu=sdom[2][1];
    int s, p, q, m;
    double acc[OH_DIMENSION];

    for (s=0,p=0; s<nspec; s++) {
        nphgram[s][n] = totalp[s];
        for (q=0; q<totalp[s]; p++,q++) {
            lorentz(eb, pbuf[p].x-xl, pbuf[p].y-yl, pbuf[p].z-zl, s, fsize, acc);
            pbuf[p].vx += acc[0];
            pbuf[p].vy += acc[1];
            pbuf[p].vz += acc[2];
            pbuf[p].x += pbuf[p].vx;
            pbuf[p].y += pbuf[p].vy;
            pbuf[p].z += pbuf[p].vz;
        }
    }

    for (s=0,p=0; s<nspec; s++) {
        nphgram[s][n] = totalp[s];
        for (q=0; q<totalp[s]; p++,q++) {
            lorentz(eb, pbuf[p].x-xl, pbuf[p].y-yl, pbuf[p].z-zl, s, fsize, acc);
            pbuf[p].vx += acc[0];
            pbuf[p].vy += acc[1];
            pbuf[p].vz += acc[2];
            pbuf[p].x += pbuf[p].vx;
            pbuf[p].y += pbuf[p].vy;
            pbuf[p].z += pbuf[p].vz;
        }
    }
```

Now we finish the double loop for species and particles in each of them. We let `nphgram[s][n]` have `totalp[s]` as its initial value at the beginning of the iteration for each species `s`, to mean that we will have `totalp[s]` in the subdomain `n` if all the particles of the species `s` stay in the subdomain. Then we call `lorentz()` to have the acceleration vector of each particle in the array `acc[3]`, whose elements are added to the velocity vector components of the particle. After this acceleration (or deceleration), the particle is moved by adding the velocity vector to the position vector.

```c
    if (pbuf[p].x<xl || pbuf[p].x>=xu ||
        pbuf[p].y<yl || pbuf[p].y>=yu ||
        pbuf[p].z<zl || pbuf[p].z>=zu) {
        m = oh_map_particle_to_neighbor(&pbuf[p].x, &pbuf[p].y, &pbuf[p].z, 
                                         ps);
        nphgram[s][n]--; nphgram[s][m]++;
        pbuf[p].nid = m;
    }
```

Now we finish the job for a particle if it is still staying in the subdomain. Otherwise, we call `oh3_map_particle_to_neighbor()` to obtain the identifier `m` of the subdomain in which the particle now resides. Then `nphgram[s][n]` is decreased by one to indicate that the particle has gone, while `nphgram[s][m]` is increased by one to represent its immigration. We also update `nid` element of the particle to show it now resides in the subdomain `m`.

```c
    if (pbuf[p].x<xl || pbuf[p].x>=xu ||
        pbuf[p].y<yl || pbuf[p].y>=yu ||
        pbuf[p].z<zl || pbuf[p].z>=zu) {
        m = oh_map_particle_to_neighbor(&pbuf[p].x, &pbuf[p].y, &pbuf[p].z, 
                                         ps);
        nphgram[s][n]--; nphgram[s][m]++;
        pbuf[p].nid = m;
    }
```
Function current_scatter()

The third function current_scatter() is given seven arguments to specify primary or secondary particles, primary or secondary subdomain and its field-array; 
\texttt{pbuf} for particle buffer; \texttt{nspc} for the number of species; \texttt{totalp} for the number of particles in each species; 
\texttt{cd} for the field-array of current density vectors; \texttt{sdn} for the size and the location of the subdomain; \texttt{ctype} to know the range in \texttt{cd} which the particles will contribute to; and \texttt{fsn} for the size of \texttt{cd}.

Then, in the local variable declaration, we get lower subdomain boundaries from \texttt{sdn} to set them into \texttt{xL} and so on, and upper boundaries to set those in the local subdomain coordinates into \texttt{uL} and so on, for the sake of conciseness. We also have local variables \texttt{w} for width of the field array \texttt{cd} and \texttt{wd} for width times depth of it to calculate the index of \texttt{cd} corresponding to the local subdomain coordinates \((x, y, z)\) by \(x + w \cdot y + wd \cdot z\).

```c
void current_scatter(struct S_particle *pbuf, int nspec, int *totalp,
                     struct current *cd, int sdn[OH_DIMENSION][2],
                     int ctye[2][3], int fsize[OH_DIMENSION][2]) {
    int xl=sdn[0][0], yl=sdn[1][0], zl=sdn[2][0];
    int xu=sdn[0][1]-xl, yu=sdn[1][1]-yl, zu=sdn[2][1]-zl;
    int w=fsize[0][1]-fsize[0][0], wd=w*(fsize[1][1]-fsize[1][0]);
    int s, p, q;
    int i, j, k;
    struct current c[2][2][2];

    First we zero-clear \texttt{cd} including the boundary planes we will send to adjacent nodes referring to \texttt{ctype}.

    for (k=ctype[0][0]; k<zu+ctype[1][0]+ctype[1][2]; k++)
        for (j=ctype[0][0]; j<yu+ctype[1][0]+ctype[1][2]; j++)
            for (i=ctype[0][0]; i<xu+ctype[1][0]+ctype[1][2]; i++)
                cd[i+w*j+wd*k].jx = cd[i+w*j+wd*k].jy = cd[i+w*j+wd*k].jz = 0.0;

    Now we start the double loop. In each iteration for a particle, we call \texttt{scatter()} to have its contribution to the current density vectors of the grid points surrounding it in the array \texttt{c[2][2][2]}, whose elements are added to the corresponding elements of \texttt{cd}.

    for (s=0,p=0; s<nspc; s++) {
        for (q=0; q<totalp[s]; p++,q++) {
            int x=pbuf[p].x-xl, y=pbuf[p].y-yl, z=pbuf[p].z-zl;
            scatter(pbuf[p], s, c);
            for (k=0; k<2; k++) for (j=0; j<2; j++) for (i=0; i<2; i++) {
                cd[[(x+i)+w*(y+j)+wd*(z+k)]].jx += c[k][j][i].jx;
                cd[[(x+i)+w*(y+j)+wd*(z+k)]].jy += c[k][j][i].jy;
                cd[[(x+i)+w*(y+j)+wd*(z+k)]].jz += c[k][j][i].jz;
            }
        }
    }
```

85
Function add_boundary_current()

The fourth function add_boundary_current() is given four arguments to specify the primary or secondary subdomain and its field-array: cd for the field-array of current density vectors; sdom for the size and the location of the subdomain; ctype to know the boundary planes in cd; and fsize for the size of cd.

In the local variable declaration, we calculate the upper boundaries \( \sigma_{x,y,z} \) of the subdomain in its local coordinates referring to sdom and set them into xu and so on. Then, to calculate the base (lowest coordinate) of the boundary planes, \( s^l_{x,y,z} \) and \( s^u_{x,y,z} \) for the planes obtained from neighbors and \( d^u_{x,y,z} \) and \( d^l_{x,y,z} \) for those to add to, and the number of lower and upper boundary planes \( n_l \) and \( n_u \), we refer to ctype elements to have the followings.

\[
\begin{align*}
    s^l_{x,y,z} &= \text{ctype}[1][1] \\
    s^u_{x,y,z} &= \text{ctype}[0][1] \\
    n_l &= \text{ctype}[1][2] \\
    n_u &= \text{ctype}[0][2] \\
    d^u_{x,y,z} &= s^l_{x,y,z} + n_l \\
    d^l_{x,y,z} &= s^u_{x,y,z} - n_u
\end{align*}
\]

That is, we suppose the planes to add to are at just inside of the planes obtained from neighbors.

```c
void add_boundary_current(struct current *cd, int sdom[OH_DIMENSION][2],
                          int ctype[2][3], int fsize[OH_DIMENSION][2]) {
    int xu=sdom[0][1]-sdom[0][0], yu=sdom[1][1]-sdom[1][0],
        sl=ctype[1][1], nl=ctype[1][2], du=sl+nl,
        su=ctype[0][1], nu=ctype[0][2], dl=sl+nu;
```

Then we call add_boundary_curr() six times for lower and upper boundary planes perpendicular to \( z \), \( y \) and \( x \) axes in this order to do the followings conceptually.

\[
\begin{align*}
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &← \\
    [s^l_z, s^u_z + n_u) \times [s^l_y, s^u_y + n_u) \times [d^l_z, d^u_z + n_l) &←
\end{align*}
\]

The operations above for a two-dimensional subdomain are illustrated in Figure 15.
Function add_boundary_curr()

The fifth function add_boundary_curr() does the followings conceptually for each current density vector component in cd for the boundary plane addition in add_boundary_current().

\[
\begin{align*}
\text{xd, xd}+nx \times \text{yd, yd}+ny \times \text{zd, zd}+nz &\leftarrow \\
\text{xd, xd}+nx \times \text{yd, yd}+ny \times \text{zd, zd}+nz + \text{xs, xs}+nx \times \text{ys, ys}+ny \times \text{zs, zs}+nz
\end{align*}
\]

```c
void add_boundary_curr(int xs, int xd, int nx, int ys, int yd, int ny,
                      int zs, int zd, int nz, struct current *cd,
                      int fsize[3][2]) {
    int w=fsize[0][1]-fsize[0][0], wd=w*(fsize[1][1]-fsize[1][0]);
    int i, j, k;
    for (k=0; k<nz; k++) for (j=0; j<ny; j++) for (i=0; i<nx; i++) {
        cd[(xd+i)+w*(yd+j)+wd*(zd+k)].jx += cd[(xs+i)+w*(ys+j)+wd*(zs+k)].jx;
        cd[(xd+i)+w*(yd+j)+wd*(zd+k)].jy += cd[(xs+i)+w*(ys+j)+wd*(zs+k)].jy;
        cd[(xd+i)+w*(yd+j)+wd*(zd+k)].jz += cd[(xs+i)+w*(ys+j)+wd*(zs+k)].jz;
    }
}
```

Function field_solve_e()

The sixth function field_solve_e() is given five arguments to specify the primary or secondary subdomain and its field-arrays; eb for the electromagnetic field-array; cd for the field-array of current density vectors; sdom for the size and the location of the subdomain; and fsizee and fsizec for the sizes of eb and cd.

In the local variable declaration, we calculate the upper boundaries \(\sigma_{x,y,z}\) of the subdomain in its local coordinates referring to sdom and set them into xu and so on. We also calculate the width and width times depth of eb and cd to set them into we, wde, wc and wdc.

```c
void field_solve_e(struct ebfield *eb, struct current *cd,
                   int sdom[OH_DIMENSION][2],
                   int fsizee[OH_DIMENSION][2], int fsizec[OH_DIMENSION][2]) {
```
Then, in the loop for $[0, \sigma_x] \times [0, \sigma_y] \times [0, \sigma_z]$, we update each electric field vector following the Maxwell’s (or Ampère’s circuital) law using $\nabla \times \mathbf{B}$ calculated by the out-of-scope function `rotate_b()` and set into `rot[3]`, and the current density vectors `cd`. Note that the constants $\varepsilon_0$ and $\mu_0$ for $\varepsilon$ and $\mu$ are assumed to have been defined somewhere in the simulation code.

```c
for (z=0; z<zu; z++) for (y=0; y<yu; y++) for (x=0; x<xu; x++) {
    rotate_b(eb, x, y, z, fsizee, rot);
    eb[x+y*we+z*wde].ex += (1/EPSILON)*((1/MU)*rot[0] + cd[x+y*wz+wdc].jx);
    eb[x+y*we+z*wde].ey += (1/EPSILON)*((1/MU)*rot[1] + cd[x+y*wz+wdc].jy);
    eb[x+y*we+z*wde].ez += (1/EPSILON)*((1/MU)*rot[2] + cd[x+y*wz+wdc].jz);
}
```

**Function field_solve_b()**

The seventh and last function `field_solve_b()` is given three arguments to specify the primary or secondary subdomain and its field-array; `eb` for the electromagnetic field-array; `sdom` for the size and the location of the subdomain; and `fsize` for the size of `eb`.

In the local variable declaration, we calculate the upper boundaries $\sigma_x, \sigma_y, \sigma_z$ of the subdomain in its local coordinates referring to `sdom` and set them into `xu` and so on. We also calculate the width and width times depth of `eb` to set them into `w` and `wd`.

```c
void field_solve_b(struct ebfield *eb, int sdom[OH_DIMENSION][2],
                   int fsize[OH_DIMENSION][2]) {
    int xu=sdom[0][1]-sdom[0][0], yu=sdom[1][1]-sdom[1][0],
        zu=sdom[2][1]-sdom[2][0];
    int we=fsizee[0][1]-fsizee[0][0], wde=we*(fsizee[1][1]-fsizee[1][0]);
    int wc=fsizec[0][1]-fsizec[0][0], wdc=wc*(fsizec[1][1]-fsizec[1][0]);
    int x, y, z;
    double rot[OH_DIMENSION];

    for (z=0; z<xu; z++) for (y=0; y<xu; y++) for (x=0; x<xu; x++) {
        rotate_e(eb, x, y, z, fsize, rot);
        eb[x+y*w+z*wd].bx += rot[0];
        eb[x+y*w+z*wd].by += rot[1];
        eb[x+y*w+z*wd].bz += rot[2];
    }
}
```
Table 2: File Dependency of Fortran Codes.

<table>
<thead>
<tr>
<th>file</th>
<th>depends on</th>
</tr>
</thead>
<tbody>
<tr>
<td>simulator</td>
<td>simulator.o sample.o oh_mod1.o ohhelp2.o 0 (l ∈ [1, L])</td>
</tr>
<tr>
<td>simulator.o</td>
<td>simulator.F90 sample.o oh_mod1.o ohhelp2.o 0 oh_stats.h+4</td>
</tr>
<tr>
<td>sample.o</td>
<td>sample.F90 oh_mod1.o ohhelp2.o 0 oh_dim.h+3 oh_stats.h+4</td>
</tr>
<tr>
<td>oh_mod3.o</td>
<td>oh_mod3.F90 oh_mod2.o oh_dim.h</td>
</tr>
<tr>
<td>oh_mod2.o</td>
<td>oh_mod2.F90 oh_mod1.o oh_dim.h</td>
</tr>
<tr>
<td>oh_mod1.o</td>
<td>oh_mod1.F90 oh_type.o oh_dim.h</td>
</tr>
<tr>
<td>oh_type.o</td>
<td>oh_type.F90</td>
</tr>
<tr>
<td>ohhelp3.o</td>
<td>ohhelp3.c ohhelp3.h ohhelp1.h oh_stats.h oh_part.h</td>
</tr>
<tr>
<td>ohhelp2.o</td>
<td>ohhelp2.c ohhelp2.h ohhelp1.h oh_dim.h oh_stats.h oh_part.h</td>
</tr>
<tr>
<td>ohhelp1.o</td>
<td>ohhelp1.c ohhelp1.h oh_dim.h oh_stats.h</td>
</tr>
</tbody>
</table>

*1 Dependence to .o files represents that a file providing a module must be compiled prior to files which use it if it is modified.
*2 If you use function aliasing.
*3 If you refer to OH_DIMENSION.
*4 If you use statistics functions.

3.12 How to make

Since the OhHelp library includes header files which may be (or is expected to be) customized to your own simulator, it should be confusing if we provide a Makefile to build a library archive which could be mistakenly assumed independent of your customization. Therefore, the distribution of OhHelp merely has samples of Makefile namely sample_F.mk and sample_c.mk to make your simulator in Fortran and C together with the library coded in C.

The sample Makefile for Fortran sample_F.mk represents the dependency shown in Table 2, while its C counterpart sample_c.mk corresponds to that shown in Table 3, providing that you choose level-L library\(^{23}\). In the sample files, it is assumed that your simulator has just two sources, sample.F90 and simulator.F90 or sample.c and simulator.c, and simulator.{F90,c} provides main routines and out-of-scope subroutines/functions used in sample.{F90,c}. It is also assumed your source files need neither of your own header files nor module files to be #include’d or use’d, although usually you should have some of them.

\(^{23}\)The tables show dependencies accurately and strictly, but sample Makefile’s have redundant (but safe) dependencies such as that ohhelp1.c depends on ohhelp3.h.
Table 3: File Dependency of C Codes.

<table>
<thead>
<tr>
<th>file</th>
<th>depends on</th>
</tr>
</thead>
<tbody>
<tr>
<td>simulator</td>
<td>simulator.o sample.o ohhelp2.o ( l \in [1, L] )</td>
</tr>
<tr>
<td>simulator.o</td>
<td>simulator.c ohhelp2.c.h ( ^1 ) oh_part.h ( ^2 ) oh_dim.h ( ^3 ) oh_stats.h ( ^4 )</td>
</tr>
<tr>
<td>sample.o</td>
<td>sample.c ohhelp3.c.h ( ^1 ) oh_part.h ( ^2 ) oh_dim.h ( ^3 ) oh_stats.h ( ^4 )</td>
</tr>
<tr>
<td>ohhelp3.o</td>
<td>ohhelp2.c ohhelp2.h ohhelp1.h oh_dim.h oh_stats.h</td>
</tr>
<tr>
<td>ohhelp2.o</td>
<td>ohhelp2.c ohhelp2.h ohhelp1.h oh_dim.h oh_stats.h oh_part.h</td>
</tr>
<tr>
<td>ohhelp1.o</td>
<td>ohhelp1.c ohhelp1.h oh_dim.h oh_stats.h oh_part.h</td>
</tr>
</tbody>
</table>

*1 If you use function aliasing.
*2 If \( L \geq 2 \).
*3 If you refer to OH_DIMENSION.
*4 If you use statistics functions.

Acknowledgments

The author thanks to Prof. Yoshiharu Omura and Prof. Hideyuki Usui who triggered the research work on OhHelp and have patiently supported the author during his snailish development. He also thanks to Dr. Yohei Miyake who kindly gave the author the first target simulator named 3D-Kempo, and to Dr. Hitoshi Sakagami who motivated the author to form his OhHelp program as a library package.
Index

Underlined number refers to the page where the specification of corresponding entry is described.

Symbols

\[ \Delta^d \] ........................................ 36, 45
\[ \Delta^d \] ........................................ 36, 45
\[ \Phi_n \] ........................................ 9, 10
\[ I_d \] ........................................... 5, 19, 23, 33
\[ \alpha \] ........................................ 36, 44
\[ \delta^d_\theta (n) \] .................................. 36, 44
\[ \delta^d_\phi (n) \] .................................. 36, 44
\[ \theta^d_\phi (f) \] .................................. 36, 44
\[ \theta^d_\Phi (f) \] .................................. 36, 44
\[ \Phi_d \] ........................................ 22, 24, 27, 37, 44
\[ \nu_d \] ........................................ 22, 24, 27, 37, 44
\[ \pi_d \] ........................................ 22, 24, 27, 37, 44
\[ \sigma_d \] ........................................ 39, 40, 46, 47, 71, 76, 78, 79, 86-88

A

add_boundary_curr() .............. 68, 76, 77, 86, 87
add_boundary_current() ............. 68, 72, 73, 77, 83, 86, 87

B

B (number of boundary condition types) ........................................ 38, 40, 45, 47
B (magnetic field) ......... 11, 12, 68, 73, 78, 83, 88
B_abs ........................................ 9
bcond .................. 38, 45, 46, 50, 52, 69-71, 79, 82
bounds .................. 37, 38, 44, 45, 50, 52, 69-71, 79, 82

C

C .................................................. 38, 40, 45, 46
C header files:
oh_dim.h ......................... 18, 67, 89, 90
oh_part.h ......................... 30, 67, 90
oh_stats.h ....................... 59, 89, 90
ohhelp1.h ......................... 11, 89, 90
ohhelp2.h ......................... 11, 89, 90
ohhelp3.h ......................... 11, 89, 90
ohhelp_c.h ................. 11, 18, 19, 24, 30, 35, 67, 79, 90
C source files:
ohhelp1.c ......................... 11, 89, 90
ohhelp2.c ......................... 11, 89, 90
ohhelp3.c ......................... 11, 89, 90
sample.c ......................... 79, 89, 90
C structs:
current ......................... 79, 82, 85-87
ebfld ....................................... 38, 40, 42, 45, 46, 50, 52, 69-71, 79, 82
S_mycommc ....................... 24, 67
S_particle ....................... 15, 20, 24, 25, 36, 39, 59, 67, 79, 84, 85
cd .................. 39, 46, 55, 57, 68, 69, 70-73, 79, 82, 83
cfields ....................... 40, 42, 46, 48, 50, 59-71, 79, 82
Fortran types:

- oh_mycomm
- oh_particle
- fsizes
- ftypes

Function aliases:

- oh_all_reduce()
- oh_allreduce_field()
- oh_bcast_field()
- oh_broadcast()
- oh_exchangeBorders()
- oh_init()
- oh_init_stats()
- oh_inject_particle()
- oh_map_particle_to_neighbor()
- oh_map_particle_to_subdomain()
- oh_max_local_particles()
- oh_print_stats()
- oh_reduce()
- oh_reduce_field()
- oh_show_stats()
- oh_statstime()
- oh_transbound()
- oh_verbose()

H

- H(n)
- helpand
- helper

I

- initialize_eb()
- initialize_particles()

J

- J (current density)

L

- Lorentz force law
- lorentz()

M

- make files:
  - sample_c.mk
  - sample_f.mk
  - malloc()
  - malloc_field_array()
  - MAXFRAC
  - maxfrac
  - maxlocalp

Maxwell's equation

MPI (standard header file)

N

- oh13_init()
- oh1_allreduce()
- oh1_broadcast()
- oh1_init()

O

- oh1_init_stats()
- oh1_reduce()
- oh1_show_stats()
- oh1_statstime()
- oh1_transbound()
- oh1_verbose()

P

- oh2_allreduce()
- oh2_bcast()
- oh2_init()
- oh2_inject_particle()
- oh2_max_local_particles()
- oh2_transbound()

Q

- oh3_allreduce()
- oh3_bcast()
- oh3_init()
- oh3_map_particle_to_neighbor()
- oh3_map_particle_to_subdomain()

R

- oh4_allreduce()
- oh4_bcast()
- oh4_init()
- oh4_inject_particle()
- oh4_max_local_particles()
- oh4_transbound()

S

- oh5_allreduce()
- oh5_bcast()
- oh5_init()
- oh5_inject_particle()
- oh5_max_local_particles()
- oh5_transbound()

T

- oh6_allreduce()
- oh6_bcast()
- oh6_init()
- oh6_inject_particle()
- oh6_max_local_particles()
- oh6_transbound()

V

- oh7_allreduce()
- oh7_bcast()
- oh7_init()
- oh7_inject_particle()
- oh7_max_local_particles()
- oh7_transbound()

W

- oh8_allreduce()
- oh8_bcast()
- oh8_init()
- oh8_inject_particle()
- oh8_max_local_particles()
- oh8_transbound()

X

- oh9_allreduce()
- oh9_bcast()
- oh9_init()
- oh9_inject_particle()
- oh9_max_local_particles()
- oh9_transbound()

Y

- oh10_allreduce()
- oh10_bcast()
- oh10_init()
- oh10_inject_particle()
- oh10_max_local_particles()
- oh10_transbound()

Z

- oh11_allreduce()
- oh11_bcast()
- oh11_init()
- oh11_inject_particle()
- oh11_max_local_particles()
- oh11_transbound()
oh3_transbound() .................................................. 18, 34, 35, 49, 50, 58, 60, 64, 67, 71, 72, 82
oh_all_reduce() (function alias) ................................. 67
oh_allreduce_field() (function alias) .......................... 67, 72, 83
oh_bcast_field() (function alias) .................................. 67, 71, 72, 82
oh_broadcast() (function alias) ................................. 67, 71, 72, 82
OH_DEFINE_STATS ............................................... 61
oh_dim.h (C header file) .................................. 18, 67, 89, 90
OH_DIMENSION .................................................. 18, 19, 31, 35, 51, 53, 69-71, 73, 75, 78, 79, 81, 84-88
oh_exchange_borders() (function alias) .................. 67, 71-73, 82, 83
oh_init() (function alias) ..................................... 67, 71, 82
oh_init_stats() (function alias) ............................... 67
oh_inject_particle() (function alias) ......................... 67
OH_LIB_LEVEL ................................................... 67, 68, 79
oh_map_particle_to_neighbor() (function alias) ....... 67, 74, 84
oh_map_particle_to_subdomain() (function alias) .......... 67
oh_max_local_particles() (function alias) ................. 67, 70, 82
oh_mod1.F90 (Fortran module file) ..................... 19, 27, 89
oh_mod2.F90 (Fortran module file) ......................... 30, 89
oh_mod3.F90 (Fortran module file) ....................... 35, 68, 89
oh_mycomm (Fortran type) .................................. 20, 31, 35, 49, 69
oh_part.h (C header file) .................................. 30, 67, 90
oh_particle (Fortran type) .................................. .... 30, 31, 32, 34, 35, 59, 69, 73, 74
oh_print_stats() (function alias) ............................ 67
oh_reduce() (function alias) .................................. 67
oh_reduce_field() (function alias) ......................... 67
oh_show_stats() (function alias) ............................. 67
oh_stats.h (C header file) .................................. 59, 89, 90
oh_stats_time() (function alias) ............................. 67
oh_transbound() (function alias) ............. 67, 71, 72, 82
oh_type.F90 (Fortran module file) ................. 20, 30, 39, 89
oh_verbose() (function alias) ............................... 67
ohhelp1 (Fortran module) .................................... 19
ohhelp1.c (C source file) .................................. 11, 89, 90
ohhelp1.h (C header file) .................................. 11, 89, 90
ohhelp2 (Fortran module) .................................... 30
ohhelp2.c (C source file) .................................. 11, 89, 90
ohhelp2.h (C header file) .................................. 11, 89, 90
ohhelp3 (Fortran module) .................................... 35, 68
ohhelp3.c (C source file) .................................. 11, 89, 90
ohhelp3.h (C header file) .................................. 11, 89, 90
ohhelp_c.h (C header file) .................................. 11, 18, 19, 24, 30, 35, 67, 79, 90
ohhelp_f.h (Fortran header file) .............. 11, 18, 67, 68, 89
<table>
<thead>
<tr>
<th>secondary family</th>
<th>secondary mode</th>
<th>secondary particle</th>
<th>secondary subdomain</th>
<th>species</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>27, 28, 29, 54, 55</td>
<td>5, 6, 7, 18, 19, 23, 25, 26, 57, 63, 65, 72, 73, 83, 84</td>
<td>5, 6, 8, 12-15, 17-20, 23, 32, 52, 61, 65, 66, 72-74, 82-85</td>
<td>4, 5, 6, 7, 12, 13, 15, 16, 19, 20, 23, 34, 47, 50-57, 61, 72-75, 78, 82-88</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14, 15, 17-20, 23, 30, 34, 71, 73-75, 82, 84, 85</td>
</tr>
<tr>
<td>standard header files:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpi.h</td>
<td>stats</td>
<td>STATS_CURRENT_SCATTERING</td>
<td>STATS_FIELD_SOLVING</td>
<td>STATS_PARTICLE_PUSHING</td>
</tr>
<tr>
<td>stdlib.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>statistics</td>
<td>stats</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>verbose</td>
<td>verbose messaging</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

94