PPM User's guide

October 12, 2011

Contents

1 Introduction

The PPM library [2, 3] defines the state of the art in middleware for distributed-memory particle-mesh simulations. It hides MPI from the application programmer by introducing an additional, transparent layer beneath the user's simulation programs (called "PPM clients"). Since PPM reduces the knowledge gap, the resulting simulations often outperform hand-parallelized codes [2, 1]. The PPM library is independent of specific applications, provided the simulation is phrased in terms of particles, meshes, or a combination of the two. PPM implements modules for adaptive domain decomposition, communication through halo layers, load balancing, particle-mesh interpolation, and communication scheduling. All of this is done transparently without participation of the user program.

Recently PPM has undergone some major design changes:

- The PPM library has been split up into two libraries. The first part is PPM core, which provides the core parallel framework consisting of abstractions for topologies, mappings, particles and meshes. The second part is PPM numerics. This new library provides all the numerical methods that were implemented based on the PPM abstractions. In this tutorial we shall concentrate on PPM core.
- Using the PPM core library has been greatly simplified. The number of arguments that must be passed to PPM routines has been reduced, topologies are now handled internally by the library.
- We spent some time reorganizing the code and generating an API reference.
- Installing PPM has become much simpler, thanks to the use of GNU Autoconf and Automake.

2 Obtaining and installing PPM

PPM is distributed in source packages for the user to compile and setup on his environment. We do not offer any pre-built packages. Go to `http://www.ppm-library.org` to download the latest version of PPM core from the "Downloads" section. PPM has been tested with Mac OS X 10.5/10.6, Ubuntu Linux (x86-64) and Gentoo Linux (x86-64). It should however work on any Unix system, provided all dependencies are present and installed. In essence PPM requires following packages to be installed:

- A FORTRAN 95 compiler. For systems with Intel CPUs we recommend the latest Intel Fortran compiler (v. 11.x). You can also use GCC's `gfortran`. Many high-performance clusters provide also vendor-specific compilers (e.g. IBM XL Fortran, Sun studio Fortran, NEC Fortran compiler).
- An MPI library. We have tested PPM with several versions of OpenMPI, but any MPI-2 compliant library should work. It is important to compile PPM with the same Fortran compiler as the MPI library. PPM can be also compiled without MPI support.
- The METIS library. METIS is a library for graph partitioning and fill-reducing matrix ordering. You can either download it from `http://glaros.dtc.umn.edu/gkhome/metis/metis/download` or from the PPM website along with the PPM core source package.
- The `make` command and functioning shell. To build PPM we make use of Makefiles that are interpreted by `make`. Along with the compilers the environment where PPM core will be compiled has to provide the standard unix shell tools and make.

After you have downloaded the PPM core `.tar.gz` source archive to your system and prepared the requirements you must unpack, configure and compile the code. In the following snipped we assume you are using OpenMPI. Please issue the `./configure --help` command to find out about how to customize PPM core:

```
tar xzvpf ppmcore1.2.tar.gz cd libppm
./configure --enable-mpi=openmpi --enable-linux --prefix=/where
    /ppm/shall/be/installed LDFLAGS=-L/directory/to/metis/lib
```

Note: For most standard installations it is not necessary to specify the compiler, as the install scripts will try to find them automatically.

the `--prefix` argument requires the given directory to be already created, otherwise the build process will fail.

If you have correctly installed all PPM dependencies and specified the correct paths in the above command the configuration process should exit after generating the Makefile:

```
configure: creating ./config.status
config.status: creating Makefile
```

The next step is to compile PPM.

2.1 Development

For development and debugging, configure can be called with the –enable-dev and the –enable-debug flags respectively.

3 Understanding the PPM parallelism

TODO: It would be useful to have a summary of what PPM actually does and how it does it. What are the different abstractions and how are they organized. Some diagrams/sketches would probably help a lot...

For now, the reader is referred to the seminal papers [2, 1].

4 Sample PPM Client Application

PPM offers a rich functionality for parallelizing the computations that can be used in PPM clients by calling the respective subroutines. A "skeleton" of a typical PPM client consists of the following steps:

1. Read program configuration from a control file
2. Initialize the PPM library
3. Create the topology and fill it in with particles :
   a) Perform the domain decomposition
   b) Generate particles
   c) Map the particles onto the topology
4. Initialize the ODE module and create an ODE mode
5. Enter the main cycle of the client :
   a) Communicate the particles on ghost layers between all processes
   b) Build neighbor lists only taking into account particles lying within cut-off range
   c) Performing main calculations:
      - Calculate approximations of the time derivatives using the ODE solver *or*
      - Perform particle–particle interactions
   d) Calculate other quantities using the results of previous step
   e) Update positions of individual particles
6. Output the results
7. Finalize the PPM Library

The first step is done by means of a simple subroutine, which reads the control file and extracts the parameter values and program configuration. The default control file has the following format: each non-blank line, except those that begin with a `'#'` character, is considered as significant and containing a pair of parameter name and value. A sample control file is given below:

```
#-------------------------------------------------------------
# Sample control file
#
# This line is a comment
#-------------------------------------------------------------
<parameter_name> = <parameter_value>
...
```

The second step is done by calling the `ppm_init` subroutine. To open the log unit a subsequent call to `ppm_io_set_unit` is required.

The topology is created using `ppm_mktopo` subroutine, which is the topology creation routine for particles. It performs the decomposition of the physical space based on the position of the particles and returns the topology id to the calling program. The subdomains are mapped onto the processors and a neighbour list is created. The topology itself can be obtained by subsequent call to `ppm_topo_get` with the topology id as the first parameter and a pointer to the topology itself. This allows access and modify the fields of the topology structure directly, however, usually there is no need to do so as `ppm_mktopo` and other library functions should provide all the necessary functionality for that.

The next step to perform is to fill this geometry with particles, which is problem-specific and may be done in a variety of ways.

The mapping the particles onto topology is performed in several steps. First `ppm_map_part_global` maps the particles onto the given topology using a global mapping (i.e. every processor communicates with every other). Then a call to `ppm_map_part_push` would push the any relevant particle data onto the send buffer that is communicated to the other processes via `ppm_map_part_send`. Other processes can access this data by subsequent calls to `ppm_map_part_pop`, which would return the particle data (positions, strength, etc) in the order it was put into the buffer.

It should be stressed out that if no geometric decomposition of the domain is required, the two previous steps should be skipped.

Problems that are solved by using PPM are ususally governed by a set of ODEs. For example, a typical transport problem of the form

$$\frac{Df}{Dt} = \frac{\partial f}{\partial t} + \nabla \cdot (\mathbf{u}f) = \mathcal{L}(f), \tag{4.0.1}$$

leads to the system of ODEs for the positions $\mathbf{x}_p$, strengths ω_p, and volumes v_p of the particles:

$$\begin{aligned} \frac{d\mathbf{x}_p}{dt} &= \mathbf{u}_p(t) \\ \frac{d\omega_p}{dt} &= v_p \mathcal{L}(f)(\mathbf{x}_p) + \frac{dv_p}{dt} f(x_p) \\ \frac{dv_p}{dt} &= v_p \nabla \cdot \mathbf{u} \end{aligned} \tag{4.0.2}$$

with $\omega_p = v_p f(\mathbf{x}_p)$. The differential operator $\mathcal{L}$ can be, e.g. a diffusion operator.

To faciliate the matter, PPM library provides subroutines that give the necessary functionality:

- `ppm_ode_init(topo_id,info)` – initializes the ode solver variable and registers the ID of the topology to be used for the ODE solver
- `ppm_ode_create_ode`, which creates an ODE solver for the given integration scheme, kick-off scheme and sets whether the solver is adaptive,
- `ppm_ode_start` – should be called before the first step of numerical integration. It check whether all solvers are ready then sets them into kick-off state thus allowing the numerical integration to begin,
- `ppm_ode_step` is the very heart of ODE solver. This routine implements Euler, Runge-Kutta 2 and Runge-Kutta 4 numerical intgration schemes. It should be called inside the main cycle of the client, to provide tnumerical approximations of governing variable for the next timestep.

The main cycle of a typical client that involves solving a system of ODEs also needs means for calculating the quantities that characterize the evolution of the flow in time. To calculate such quantities at particle's positions, e.g. concentrations of chemical species, flow vorticities . . . , particle strength's of the neighbouring particles are usually required. In the PPM library such data can be obtained by means of *neighbour lists*. To avoid boundary effects, especially when run on parallel processors, particles that are close to the boundary of the (sub)domain get additional neighbours from the particles lying in adjacent subdomain within some distance (called *cutt-off distance*) that implemented as particles on ghost layers. Each subdomain that is assigned to a separate processor has several ghost layers, each for one of his neighbours on other processors.

Having computed the values of all physical quantities and updated the positions of all particles means that one cycle of the main cycle of the client is finished and, after storing the results of the calculations, the client is ready to proceede to the next step.

Main cycle of a client that solves a problem involving many particle-to-particle interactions usually requires many communication between all the processors. To reduce the communication overhead, . . .

Each step of the simple will be discussed in more detail.

4.1 Typical variable names

`Npart`: number of real particles
`Mpart`: number of (real+ghosts) particles
`xp`: array of particles' positions
`wp`: array of particles' strengths

4.2 Reading the control file

In this simple case, control file governs these main aspects of the simulation:

- grid stepping – defined using the parameter:

```
Ngrid = 43,43,43
```

- name and how often an output file will be written:

```
freqoutput = 100000
outputfile = pseoutput
outputfmt  = UNFORMATTED
```

- settings for continuing the simulation (restart):

```
restart     = FALSE
restartfile = pse_restart00000011.rst
freqrestart = 1000
```

- problem parameters, e.g. number of spatial dimensions, size of computational domain, number of particle species, isotropic diffusion constants for all species

```
dimensions  = 3
domain      = 0.0, 0.0, 0.0, 4.0, 4.0, 4.0
species     = 1
isodiff     = 0.03
```

- PSE method parameters like the kernel function to use, core radius of the kernel, cut-off distance of particle-particle interaction, etc

These sections are ones that should be necessary to perform almost any simulation using PPM library. In general, the control file is fine-tailored for the specific aspects simulation and may contain some additional problem-specific sections. Next step is to load all this data into PPM.

4.3 Initializing PPM

Before calling any of the PPM routines, a call to `ppm_init` is mandatory. It takes several parameters:

- `in` dimensionality of the problem,
- `in` precision,
- `in` cutoff tolerance,
- `in` MPI communicator,
- `in` flag indicating a debug run,
- `out` parameter to return the status of the call,
- `out` handle to log unit

For example, consider this FORTRAN 95 code :

```
1 !----------------------------------------
2 !  Initialise the ppm library
3 !----------------------------------------
4 tolexp = INT(LOG10(EPSILON(cutoff)))+10
5 CALL ppm_init(ndim,MK,tolexp,comm,debug,info,ppm_log_unit)
6 IF (info .NE. 0) THEN
```

```
7   CALL pwrite(rank,'pse_init','Failed to initialize ppm library
        .',info)
8   GOTO 9999
9  ENDIF
10 !--------------------------------------
11 !  Set ppm log file unit
12 !--------------------------------------
13 CALL ppm_io_set_unit(6,0,ppm_log_unit,info)
14 IF (info .NE. 0) GOTO 9999
15 ...
```

5 Basic routines and data structures

A topology is defined as a partition of the computational domain into several subdomains and the assignment of each subdomain to a processor. The data structures of the computational elements, particles and meshes, are constructed relative to one given topology. Ideally, the goal is for this topology to be built so as to equi-distribute the computational load amongst all the processors and to minimize the communication overhead between the compute nodes.

PPM does not yet fully achieve this goal but it is being developped in that direction. To

5.1 Domain decomposition

5.1.1 Distinguish: Mesh / Field

Meshes are created on subdomains. Therefore the subs extents must be compatible with the mesh spacing in all spatial dimensions. This means: the sub extent must be an integer multiple of the mesh spacing.

User defines the number of mesh points (not cells!) of the global mesh in all dimensions. The mesh spacing is then computed internally from this number and the extent of the physical domain. There is always a grid point placed right ON the boundary of the computational domain (PUT A SKETCH HERE).

Due to above-mentioned compatibility contraint, sub boundary faces always collocate with mesh planes (lines in 2D). The mesh points ON these planes belon to BOTH subs. This seemingly unneccessary duplication of points is motivated by:

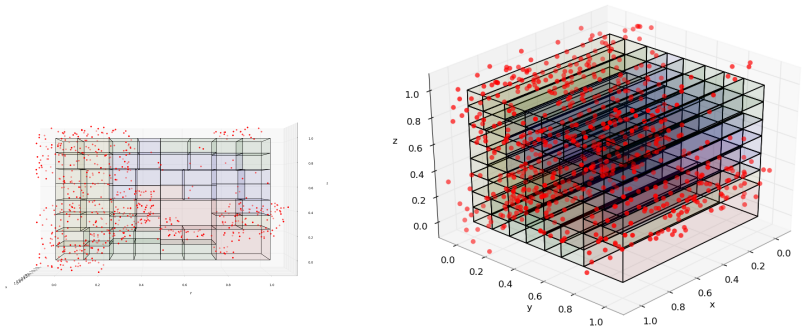

Figure 5.1.1: Example of slab domain decomposition based on particles

1. provide a convergence criterion for the Multigrid
2. in the case of periodic outer B.C. the situation is consistent (the points coincide around the boundary.
3. internal boundaries (between subs) and external ones (comput. domain) can be treated the same and the solvers do not need to care about where there subs are locates.
4. M'4 remeshing needs a ghostlayer of size 1 on all sides. Otherwise it would have needed size 1 on one side and 2 on the other. Asymmetric ghost layers are a pain for the mapping functions.

Circumvent by: not computing the same point twice (or compute and use as convergence detection).

When doing a field or field ghost mapping, the values on the duplicated mesh nodes are actually exchanged (replaced with the value on the same mesh point on the other sub or added for the ghost put). This always applies to all internal sub boundaries and in case of periodic boundary conditions also for the mesh points right on the boundary of the computational domain. (of course only in periodic directions). This whole paragraph is maybe too messy and there should be a better way to describe this. Maybe also add a figure.

5.1.2 Particle Based

The domain decomposition can be driven by the positions of the particles. In that case PPM creates subdomains such that each of them contains a similar number of particles. To better equidistribute the computational load between the processors, a cost-per-particle can be given as an input argument. PPM will then adjust the sizes of the subdomains such that the cost is equidistributed.

```
CALL ppm_mktopo(topoid,xp,np,decomp,assig,min_phys,max_phys,
    bcdef,ghostsize,cost,info)
```

5.1.3 Field Based

If a mesh is provided in the argument list, then the subdomains are created such that their boundaries coincide with the grid lines of the mesh. This is necessary if one wants to use PPM's data structure for meshes.

```
CALL ppm_mktopo(topoid,meshid,xp,np,decomp,assig,min_phys,
    max_phys,bcdef,ghostsize,cost,istart,ndata,nm,info)
```

5.1.3.1 Different decompositions

the fifth parameter of the ppm_mktopo subroutine provides a variety of methods for domain decomposition, which are listed below :

- ppm_param_decomp_pruned_cell
- ppm_param_decomp_tree
- ppm_param_decomp_bisection

- ppm_param_decomp_xpencil
- ppm_param_decomp_ypencil
- ppm_param_decomp_zpencil
- ppm_param_decomp_xy_slab
- ppm_param_decomp_xz_slab
- ppm_param_decomp_yz_slab
- ppm_param_decomp_cuboid
- ppm_param_decomp_user_defined

5.1.3.2 Load balancing

5.2 Meshes

Example: defining a topology containing a mesh and allocating a field array with the right size (accounting for ghost layers).

```
1 CALL ppm_mktopo(topoid,meshid,xp,np,decomp,assig,min_phys,
    max_phys,bcdef,ghostsize,cost,istart,ndata,nm,info)
2 ALLOCATE(field_wp(nspec,(1-ghostsize(1)):(ndata(1,1)+ghostsize
    (1)),(1-ghostsize(2)):(ndata(2,1)+ghostsize(2)),1))
```

5.3 Mapping between particles and processors

Redistributes particles to their corresponding processor (as defined by the topology).

5.3.1 Global mapping

All-to-all communication.

```
CALL ppm_map_part_global(topoid,xp,Npart_old,info) ! positions
CALL ppm_map_part_push(wp1,Npart_old,info) ! strengths
CALL ppm_map_part_push(wp2,dim2,Npart_old,info) ! vector
    property
CALL ppm_map_part_send(Npart_old,Npart_new,info) ! send
CALL ppm_map_part_pop(wp2,dim2,Npart_old,Npart_new,info)
CALL ppm_map_part_pop(wp1,Npart_old,Npart_new,info)
CALL ppm_map_part_pop(xp,ndim,Npart_old,Npart_new,info)
```

5.3.2 Partial mapping

Communication restricted to neighboring processors. This is much faster and is generally used to update the mappings when particles have moved (this assumes that no particle has moved further than the width of one subdomain).

```
CALL ppm_map_part_partial(topoid,xp,Npart_old,info) ! positions
CALL ppm_map_part_push(wp1,Npart_old,info) ! strengths
CALL ppm_map_part_push(wp2,dim2,Npart_old,info) ! vector
    property
CALL ppm_map_part_send(Npart_old,Npart_new,info) ! send
CALL ppm_map_part_pop(wp2,dim2,Npart_old,Npart_new,info)
CALL ppm_map_part_pop(wp1,Npart_old,Npart_new,info)
CALL ppm_map_part_pop(xp,ndim,Npart_old,Npart_new,info)
```

5.4 Ghost layers - mappings

5.4.1 Description of ghost layers

5.4.2 Communications to and from ghost layers

Explain how the data in the ghost layers is communicated across processors

5.4.3 Ghost mappings for meshes

```
!-------------------------------------------------------------
!update the ghosts of velocity and vorticity fields
!-------------------------------------------------------------
CALL ppm_map_field_ghost_get(topoid,mesh_id,ghostsize,info)
CALL ppm_map_field_push(topoid,mesh_id,field_vorticity,vdime,
    info)
CALL ppm_map_field_push(topoid,mesh_id,field_velocity,vdime,
    info)
CALL ppm_map_field_send(info)
CALL ppm_map_field_pop(topoid,mesh_id,field_velocity,vdime,
    ghostsize,info)
CALL ppm_map_field_pop(topoid,mesh_id,field_vorticity,vdime,
    ghostsize,info)
```

5.4.4 Ghost mappings for particles

```
!-------------------------------------------------------------
!update the properties wp1 and wp2 on the ghost particles
!-------------------------------------------------------------
CALL ppm_map_part_ghost_get(topoid,xp,ndim,Npart,isymm,cutoff,
    info)
CALL ppm_map_part_push(wp1,Npart,info)
CALL ppm_map_part_push(wp2,Npart,info)
CALL ppm_map_part_send(Npart,Mpart,info)
CALL ppm_map_part_pop(wp2,Npart,Mpart,info)
CALL ppm_map_part_pop(wp1,Npart,Mpart,info)
CALL ppm_map_part_pop(xp,ndim,Npart,Mpart,info)
```

5.4.5 Other routines

ppm_map_type_isactive: query the current map type.

This can be used to check if we are currently mapping ghost_gets and can skip the the ghost_get itself only issuing push/send/pops. Useful when particles have not moved.

5.5 Neighbour lists

5.5.1 Homogeneous neighbour lists

Here the variable cutoff is a real. All the particles have the same cutoff radius.

```
CALL ppm_neighlist_vlist(topoid,xp,Mpart,cutoff,skin,symmetry,
    vlist,nvlist,info)
```

5.5.2 Inhomoneneous neigbour lists (multiresolution)

Here the variable cutoff is an array (every particle has its own cutoff radius)

```
CALL ppm_inl_vlist(topo_id,xp,Npart,Mpart,cutoff,skin,symmetry,
    ghostlayer,info,vlist,nvlist)
```

5.5.3 Cross-set inhomoneneous neigbour lists (multiresolution)

Construct the list of neighbours of one set of particles (blue particles) for each particle of another set (red particles). Here the variable cutoff is an array (every particle has its own cutoff radius)

```
CALL ppm_xset_inl_vlist(topo_id,xp_red,Np_red,Mp_red,xp_blue,
    Np_blue,Mp_blue,cutoff_blue,skin,ghostlayer,info,vlist,
    nvlist)
```

5.6 Boundary conditions

The topology data structure contains information about the boundaries of each subdomain. Each subdomain has 4 or 6 boundaries (in 2d and 3d, respectively), which can be independently internal or external. TODO: describe this in more detail? (see source code of the data structure for more).

Example: enforcing a Dirichlet boundary condition on a field.

```
!Boundary condition on phi
dirichlet_value = -1._MK
DO isub=1,topo%nsublist
  isubl = topo%isublist(isub)
    !Check whether this subdomain has external boundaries
  ! if so, applies dirichlet boundary condition

  !west boundary?
  IF (topo%subs_bc(1,isubl) .NE. 0) THEN
    field_wp(1,1,:,:,isub) = dirichlet_value
  ENDIF
  !east boundary?
  IF (topo%subs_bc(2,isubl) .NE. 0) THEN
    field_wp(1,ndata(1,isubl),:,:,isub) = dirichlet_value
  ENDIF
  !south boundary?
  IF (topo%subs_bc(3,isubl) .NE. 0) THEN
    field_wp(1,:,1,:,isub) = dirichlet_value
```

```
  ENDIF
  !north boundary?
  IF (topo%subs_bc(4,isubl) .NE. 0) THEN
    field_wp(1,:,ndata(2,isubl),:,isub) = dirichlet_value
  ENDIF
  !bottom boundary?
  IF (topo%subs_bc(5,isubl) .NE. 0) THEN
    field_wp(1,:,:,1,isub) = dirichlet_value
  ENDIF
  !top boundary?
  IF (topo%subs_bc(6,isubl) .NE. 0) THEN
    field_wp(1,:,:,ndata(3,isubl),isub) = dirichlet_value
  ENDIF
END DO
```

6 Numerical methods

6.1 ODE solver

NOTE: this section's tutorial uses the old PPM datastructure and should be updated to the new one.

Example: we want to integrate

$$\frac{du_p}{dt} = -\lambda u_p \tag{6.1.1}$$

on the particles using the 4th order Runge-Kutta of the ODE suite.

We'll assume that we have `np` particles with positions `xp` and values `up` already allocated and initialized.

6.1.1 Local variables

```
REAL(mk), DIMENSION(:,:), POINTER :: dup ! du/dt on particles
REAL(mk), DIMENSION(:,:), POINTER :: bfr ! storage space for
    the stages
REAL(mk), DIMENSION(4) :: time ! time things
INTEGER :: istage ! stage counter
INTEGER :: nstages ! number of stages
INTEGER :: bfrsz ! size of the buffer "bfr"
INTEGER :: scheme ! which scheme to use
INTEGER :: odeid ! handle on the solver
LOGICAL :: adapt ! use adaptive time step
INTEGER :: lda ! leading dimension of our mode
INTEGER, EXTERNAL :: MyRHS ! your implementation of the RHS
```

6.1.2 Initialize the ODE Suite

```
!-----------------------------------------------
! Initialize the Ode solver
!-----------------------------------------------
CALL ppm_ode_init (info)
```

6.1.3 Create Mode

```
scheme = PPM_PARAM_ODE_SCHEME_RK4 ! we want the 4th order RK
    scheme
odeid = -1 ! let the PPM choose an ID for us
adapt = .FALSE.! don't need adaptive time stepping
lda = 2
```

```
!-----------------------------------------------
! Create the mode
!-----------------------------------------------
CALL ppm_ode_create_ode(odeid, bfrsz, nstages, scheme, scheme,
    adapt, info)
```

6.1.4 Allocate space for the stages

```
ALLOCATE(bfr(bfrsz*lda,np))
```

6.1.5 Set the time

```
dt = 0.1
time(1) = 0.0
! set the start time
time(2) = 1.0
! set the end time
time(3) = 0.0
! set the current time
time(4) = dt
! set the time step size
```

6.1.6 Start the ode solver

Once all the modes have been created, start the ode solver.

```
CALL ppm_ode_start(info)
```

6.1.7 Start the time integration

```
DO WHILE(.NOT.ppm_ode_alldone(info))
DO istage=1,nstages
CALL ppm_ode_step(odeid, xp, up, dup, lda, np, & & bfr, istage,
     time, MyRHS, info=info)
!-- say particles move, then we need to map after each stage
maptype = ppm_param_map_partial
CALL ppm_map_part(xp,3,np,mpart,topo_id,maptype,info)
maptype = ppm_param_map_push
CALL ppm_map_part(up,lda,np,mpart,topo_id,maptype,info)
CALL ppm_map_part(dup,lda,np,mpart,topo_id,maptype,info)
!-- now have the ode suite map the stages
CALL ppm_ode_map_push(odeid,bfr,lda,np,mpart,info)
!-- send maptype = ppm_param_map_send
CALL ppm_map_part(dup,lda,np,mpart,topo_id,maptype,info)
!-- pop in the reverse order
CALL ppm_ode_map_pop(odeid,bfr,lda,np,mpart,info)
maptype = ppm_param_map_pop
CALL ppm_map_part(dup,lda,np,mpart,topo_id,maptype,info)
CALL ppm_map_part(up,lda,np,mpart,topo_id,maptype,info)
CALL ppm_map_part(xp,3,np,mpart,topo_id,maptype,info)
END DO
END DO
CALL ppm_ode_finalize(info)
```

6.1.8 The right-hand side

A possible implementation of the function MyRHS

```
FUNCTION MyRHS(vxp, vup, vdup, vdime, vnp, rpack, ipack, lpack,
     info)
USE myGlobalData
!-- Arguments
INTEGER, INTENT(in) :: vdime, vnp
REAL(mk),DIMENSION(:,:),POINTER :: vxp, vup, vdup
REAL(MK),DIMENSION(:,:), POINTER,OPTIONAL :: rpack
INTEGER, DIMENSION(:,:), POINTER,OPTIONAL :: ipack
LOGICAL, DIMENSION(:,:), POINTER,OPTIONAL :: lpack
INTEGER, INTENT(inout) :: info INTEGER :: MyRHS
!-- Local variables
INTEGER :: p
!-- Compute the right-hand side
! assuming the parameter REAL(mk), DIMENSION(2) :: lambda
! is specified in the module myGlobalData
DO p=1,vnp
  dup(1,p) = -lambda(1) * up(1,p) dup(2,p) = -lambda(2) * up(2,
      p)
END DO
!-- bogus return value MyRHS = 123456
RETURN
END FUNCTION MyRHS
```

6.2 Particle remeshing

6.2.1 mesh-to-particle interpolation (ppm_interp_m2p)

```
CALL ppm_interp_m2p(topoid,meshid,xp,np,wp,1,kernel,ghostsize,
    field_wp,info)
```

NOTE: there is no need to update the ghost layers (for field_wp) prior to this call.

6.2.2 particle-to-mesh interpolation (ppm_interp_p2m)

```
CALL ppm_interp_p2m(topoid,meshid,xp,np,wp,1,kernel,ghostsize,
    field_wp,info)
```

NOTE: there is no need to update the ghost particles (for xp) prior to this call. ppm_interp_p2m uses the ghost_put mapping routines internally.

6.3 Particle–particle interactions

7 I/O and visualisation

7.1 Parallel I/O

ppm_io, ASCII, binary, etc...

Not many formats are supported "out-of-the-box", but some are under development and should be available soon (e.g. VTK, HDF5).

7.2 Tools

[Under development] Visualisation of the domain decomposition, positions of the particles (ghost and/or real particles) and of the ghost layers (e.g. see figure 5.1.1).

8 Data structure for particles

In the "experimental" *dcops* branch of PPM, there is a derived type for particles (ppm_t_particles) and a collection of routines that perform some standard-/basic operations on these particles. The aim is to make it easier to write clients for the library, without having to compromise on flexibility nor on computational efficiency.

The different fields of the data structure are presented below. Note that most of these fields can remain empty (nullified pointers) such that the memory overhead is negligible. Some of the subroutines that can be used with this data structure are explained briefly. The main idea is that they check a bunch of pre-conditions and exit with an error message if any of these conditions is not fulfilled. Otherwise, it performs a task (usually passing data pointers to one or several low-level PPM subroutines), updates some book-keeping variables in the data structure (post-conditions) and exits.

8.1 Data

- xp: array of positions
- wpi: array of pointers to integer properties
- wps: array of pointers to scalar properties
- wpv: array of pointers to vector properties
- ops: container for DC operators
- nvlist: number of neighbours
- vlist: Verlet lists

8.1.1 Sizes

- Npart: local number of real particles
- Mpart: local number of real+ghosts particles
- nwpi: number of integer properties
- nwps: number of scalar properties
- nwpv: number of vector properties

8.1.2 Properties

The pointers in wpi, wps and wpv point to derived types that hold one property that is being carried by the particles. These derived types (ideally, there should be one for each data type that PPM supports....) contain:

- vec: array that contains the actual data (it is 1d for scalar properties and 2d for vector ones)
- name: the name of that property (optional, but useful for keeping track of what is what)
- has_ghosts: boolean that is true only when the ghost values for this property are up-to-date
- is_mapped: boolean that is true only when there is a one-to-one mapping between particles and this property
- map_parts: if true (default), then the mapping between this property and the particles is kept over time (e.g. during a partial mapping or after interpolation)
- map_ghosts: if true (default), then the ghost values for this property are kept up-to-date (whenever a ghost mapping is called). Otherwise, this property is not communicated.

8.2 Variables and parameters

8.2.1 Neighbour lists

- cutoff
- skin
- isymm

8.2.2 Adaptive particles

- rcp_id: index where cutoff radii are stored
- D_id: index where preferred distance D is stored
- Dtilde_id: index where local monitor function is stored
- adapt_wpid: index where the field on which the monitor function depends is stored
- G_id: index where the anisotropy tensor is stored

8.2.3 Level sets

- level_id: index where the level function is stored

8.3 Logical flags/markers

- areinside: all particles are inside the computational box
- active_topoid: active topology for the particles
- has_ghosts: ghost particles have been fetched
- ontopology: all particles are on their corresponding processor (ie. Npart is correct)
- neighlists: neighbour lists are up-to-date
- adaptive: particles are adaptive particles
- anisotropic: particles are anisotropic
- level_set: particles carry a level set function

8.4 Routines

Here we assume that topoid refers to the id of a previously defined topology and

```
TYPE(ppm_t_particles), POINTER :: Particles
```

8.4.1 apply_bc

Apply boundary conditions (wraps particles around the domain for periodic boundary conditions and deletes particles on the other side of freespace boundary conditions. Does not include other types of BC, yet.)

```
particles_apply_bc(Particles,topoid,info)
```

8.4.2 mapping_global

```
particles_mapping_global(Particles,topoid,info)
```

8.4.3 mapping_partial

```
particles_mapping_partial(Particles,topoid,info)
```

8.4.4 mapping_ghosts

```
particles_mapping_ghosts(Particles,topoid,info)
```

- Needs particles to be mapped onto the topology
- Update ghosts only for properties with the flag "map_ghosts" set to true (default)
- Skip the ghost_get if the positions and the cutoffs of the particles haven't changed (i.e. if Particles%has_ghosts is true)
- Skip the ghost_push / ghost_pop for properties that already have their ghost up-to-date (e.g. if Particles%wps(prop_id)%has_ghosts is true)

8.4.5 Create a new property

For a scalar property:

```
particles_allocate_wps(Particles,prop_id,info,name=example)
```

for a vector property with 7 dimensions:

```
particles_allocate_wpv(Particles,prop_id,7,info,name=example)
```

If prop_id = 0, then a new id is returned. Otherwise, the existing property with that id is overwritten.

8.4.6 Compute neighbor lists

```
particles_neighlists(Particles,topoid,info)
```

Some options have not yet been implemented, e.g. symmetric interactions, but that will not be difficult.

8.4.7 Access to the variables

To access the arrays where the data is stored, use

```
wp => get_wpi(Particles, prop_id)
```

for integers (wp needs to be an INTEGER, DIMENSION(:), POINTER)

```
wp => get_wps(Particles, prop_id)
```

for scalars (wp is a REAL(8), DIMENSION(:), POINTER)

```
wp => get_wpv(Particles, prop_id)
```

for vector properties (wp is then a REAL(8), DIMENSION(:,:), POINTER).

If this property is mapped onto the particles, this will return an array of length Particles%Npart. Else, it will print out an error message and return the NULL pointer. If the ghost values are also needed, use

```
wp => get_wps(Particles, prop_id,with_ghosts=.TRUE.)
```

If the ghost values for this property are available, this will return an array of length Mpart, else it will print out an error message and return the NULL pointer (which will probably crash the code a moment later)

Once these arrays are no longer needed, it is strongly advised to use the set_wpi/set_wps/set_wpv functions (which do a bookkeeping job).

If the values have been changed:

```
wp => set_wps(Particles, prop_id)
```

If the values have been changed but the ghost values have also been changed (to their correct values, such that no further communication is required), do:

```
wp => set_wps(Particles, prop_id,ghosts_ok=.TRUE.)
```

If the values have been accessed but not changed, use:

```
wp => set_wps(Particles, prop_id,read_only=.TRUE.)
```

There are corresponding functions for particles' positions:

```
xp => get_xp(Particles)
```

Note that the following call

```
xp => set_xp(Particles)
```

notifies the data structure that the particles have moved and updates the state variables accordingly. For example, the particles are no longer assumed to be within the bounds of the computational domain, and the neighbor lists, the ghosts, the DC operators as now flagged as being wrong. If the array xp has only be used for reading, one should of course call

```
xp => set_xp(Particles,read_only=.TRUE.)
```

instead.

Note that all this data can still be accessed directly:

```
Particles%xp(ip)
Particles%wps(prop_id)%Vec(ip)
Particles%wpv(prop_id)%Vec(3,ip)
```

or passed to an existing PPM subroutine. Since in that case, no checks are performed, this should be considered as a hacked and used with caution.

8.4.8 Manual override

Some routines are here to do only a book keeping job. For example if one wants to tell the library that the particles cutoffs have changed, or that the particles have moved, or that their numbers have changed, one can call

updated_cutoff, updated_positions or updated_nb_part.

The internal variables will then be updated accordingly.

8.4.9 Discretization-Corrected operators

The use of DC operators to approximate derivatives on scattered collocation points is made very easy through this data structure (see example below)

8.5 Usage examples

Let's solve a PDE of the form

$$\frac{\partial w}{\partial t} = \frac{\partial^2 w}{\partial x^2} - \frac{\partial w}{\partial y} + 2\frac{\partial^4 w}{\partial x \partial y^3}$$

in a periodic box.

```
1 !Load necessary modules
2 USE ppm_module_particles
3 USE ppm_module_dcops
4 USE ppm_module_io_vtk
5
6 !Create a topology based on geometry (ie. define a simple box)
7 topoid = 0
8 call ppm_mktopo(topoid,decomp,assig,min_phys,max_phys,bcdef,
      cutoff,cost,info)
9
10 !Initialize N particles inside the computational domain
```

```
11 call particles_initialize(Particles,np_global,info,
       ppm_param_part_init_cartesian,topoid)
12
13 !Move them around randomly by an amount of 0.15*h
14 allocate(disp(ndim,Particles%Npart))
15 call random_number(disp)
16 disp=0.15_mk*Particles%h_avg*disp
17 call particles_move(Particles,disp,info)
18
19 !Put particles back into the box using periodic boundary
       conditions
20 call particles_apply_bc(Particles,topoid,info)
21
22 !Define particles cutoff
23 Particles%cutoff = Particles%h_avg * 3.3_mk
24
25 !Global mapping
26 call particles_mapping_global(Particles,topoid,info)
27
28 !Allocate and initialize one property on these particles (using
        an external function f0_fun)
29 wp_id=0
30 call particles_allocate_wps(Particles,wp_id,info)
31 wp => Get_wps(Particles,wp_id); xp => Get_xp(Particles)
32 FORALL(ip=1:Particles%Npart) wp(ip) = f0_fun(xp(1:ndim,ip),ndim
       )
33 wp => Set_wps(Particles,wp_id); xp => Set_xp(Particles,
       read_only=.TRUE.)
34
35 !Get ghost particles and update values for wp
36 call particles_mapping_ghosts(Particles,topoid,info)
37
38 !Compute neighbor lists
39 call particles_neighlists(Particles,topoid,info)
40
41 !Allocate another field dwp
42 dwp_id=0
43 call particles_allocate_wps(Particles,dwp_id,info)
44
45 !Define a DC operator to compute the RHS of the PDE
46 nterms=3
47 allocate(degree(nterms*ndim),coeffs(nterms),order(nterms))
48 degree = (/2,0, 0,1, 1,3/)
49 coeffs = (/1.0_mk, -1._mk, 2._mk/)
50 order = (/2, 2, 2/)
51 eta_id = 0
52 call particles_dcop_define(Particles,eta_id,coeffs,degree,order
       ,nterms,info,name="my_rhs")
53
54 !Compute the DC operator
55 call particles_dcop_compute(Particles,eta_id,info)
56
57 DO WHILE (t < t_end)
58
```

```
59   !Evaluate the DC operator using wp as input data and store
         the result in dwp
60   call particles_dcop_apply(Particles,wp_id,dwp_id,eta_id,info)
61
62   !Update wp using a forward Euler scheme
63   wp => Get_wps(Particles,wp_id)
64   dwp => Get_wps(Particles,dwp_id)
65   FORALL(ip=1:Particles%Npart)
66     wp(ip) = wp(ip) + dt * dwp(ip)
67   END FORALL
68   wp => Set_wps(Particles,wp_id)
69   dwp => Set_wps(Particles,dwp_id,read_only=.TRUE.)
70
71   !Update ghost values for wp (without recomputing the ghost
         mappings)
72   call particles_mapping_ghosts(Particles,topoid,info)
73 ENDDO
74
75 !Printout the result in VTK format, to be read directly within
       e.g. Paraview
76 call ppm_vtk_particle_cloud('mydatafile',Particles,info)
77
78 !Free memory from this DC operator
79 call particles_dcop_free(Particles,eta_id,info)
80
81 !Free memory from the particles
82 call ppm_alloc_particles(Particles,N,ppm_param_dealloc,info)
```

9 Examples

Bibliography

[1] I. F. Sbalzarini. Abstractions and middleware for petascale computing and beyond. *Intl. J. Distr. Systems & Technol.*, 1(2):40–56, 2010.

[2] I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos. PPM – a highly efficient parallel particle-mesh library for the simulation of continuum systems. *J. Comput. Phys.*, 215(2):566–588, 2006.

[3] I. F. Sbalzarini, J. H. Walther, B. Polasek, P. Chatelain, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos. A software framework for the portable parallelization of particle-mesh simulations. *Lect. Notes Comput. Sc.*, 4128:730–739, 2006.