DISPATCH Documentation

Åke Nordlund

Mar 24, 2020

Table of contents:

CHAPTER 1

Overview

DISPATCH

- is a code framework, rather than just another MHD/HD code
- is a system for task based computing with, in principle, unlimited scaling
- supports different kinds of (co-existing) tasks (e.g., cell-, ray- & particle-based), using HD, MHD, RMHD, non-ideal MHD, and / or particle-in-cell solvers
- can support and boost the performance of both existing and new solvers
- relieves solvers from dealing with MPI communication and OpenMP parallelization
- makes it trivial to implent new solvers, which are required to perform only two tasks:
    1. choose a time step size, based on local variables
    2. update its state, given guard cell or other neighbor information, provided by the framework

1.1 Presentations

- Short presentation at From Stars to Planets II, Gothenburg 2019
- Invited talk at Zooming in on Star Formation, Nafplio, Greece 2019
- Invited talk at Origins of Solar Systems, Gordon Research Conference 2019
- Invited talk at Modelling High-Mass Stellar Feedback, Tubingen 2020

1.2 Publications

- Pan, L., Padoan, P., & Nordlund, Å., "Inaccuracy of Spatial Derivatives in Riemann Solver Simulations of Supersonic Turbulence" , The Astrophysical Journal 876, 90 (2019). arXiv:1902.00079

- Pan, L., Padoan, P., & Nordlund, Å., "The Probability Distribution of Density Fluctuations in Supersonic Turbulence" , arXiv e-prints arXiv:1905.00923 (2019). arXiv:1905.00923
- Popovas, A., Nordlund, Å., & Ramsey, J. P., "Pebble dynamics and accretion on to rocky planets - II. Radiative models" , Monthly Notices of the Royal Astronomical Society 482, L107 (2019). arXiv:1810.07048
- Popovas, A., Nordlund, Å., Ramsey, J. P., & Ormel, C. W., "Pebble dynamics and accretion on to rocky planets - I. Adiabatic and convective models" , Monthly Notices of the Royal Astronomical Society 479, 5136 (2018). arXiv:1801.07707
- Ramsey, J. P., Haugbølle, T., & Nordlund, Å., "A simple and efficient solver for self-gravity in the DISPATCH astrophysical simulation framework" , Journal of Physics Conference Series 1031, 012021 (2018). arXiv:1806.10098
- Nordlund, Å., Ramsey, J. P., Popovas, A., & Küffmeier, M., "DISPATCH: a numerical simulation framework for the exa-scale era - I. Fundamentals" , Monthly Notices of the Royal Astronomical Society 477, 624 (2018). arXiv:1705.10774

1.3 License

DISPATCH is licensed under a BSD 3-Clause license:

```
Copyright (c) 2019 Åke Nordlund, Jon Ramsey, Andrius Popovas, Troels Haugbølle,␣
↪Michael Kuffmeier.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are␣
↪permitted
provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of
conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this␣
↪list of
conditions and the following disclaimer in the documentation and/or other materials␣
↪provided
with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be␣
↪used to
endorse or promote products derived from this software without specific prior written␣
↪permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY␣
↪EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF␣
↪MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT␣
↪HOLDER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS␣
↪OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND␣
↪ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING␣
↪NEGLIGENCE OR
```

(continues on next page)

(continued from previous page)

1.4 Collaborations

We welcome collaborations, in which we give access to the development version of DISPATCH, or provide support in connection with use of the public repository. Such collaborations are expected to result in contributions in the form of added solvers and/or experiment setups. After some time, such additions are expected to migrate to the public repository.

In most cases such collaborative development efforts run smoothest if forks of the development repository (or possibly of the public repository) are used, with pull requests used to merge validated and stable methods into the main repository.

In cases where our contributions to use of DISAPTCH are significant, those of us that participate also may expect co-authorship.

CHAPTER 2

User Guide

2.1 Installing

To clone a working copy of the code from Bitbucket, do

```
git clone https://aanordlund@bitbucket.org/aanordlund/dispatch.git
```

To work with GIT behind a firewall – including cloning – see the link below.

To contribute back to the repository, use pull requests from a branch, or (better) setup a fork on Bitbucket (search for "Forking a repository" on Bitbucket Support), and make pull requests from the fork.

To get direct write access (mainly for core developers and collaborators), setup an account on Bitbucket if you don't have one, and ask to be added to the access list. Then use the URL `git@bitbucked.org:aanordlund/dispatch` to clone or fork.

2.1.1 GIT behind a firewall

To use GIT with bitbucket on a cluster with a firewall you can use git over SSH, and "tunnel" port 2222 on the remote machine to port 22 at bitbucket.org, via your laptop. You also need to have an account at bitbucket.org (useful in any case); cf. separate page. When this is arranged, do

```
ssh -R 2222:bitbucket.org:22 your_login@host_behind_firewall
```

From the remote host point of view, the repository is available on the local port (2222), and therefore the clone command look like so:

```
git clone ssh://git@localhost:2222/aanordlund/dispatch.git
```

Once cloned, `git pull` and other GIT commands that need access to the repository will work, as long as the SSH tunnel is connected.

If you want avoid having to add the extra options to the SSH command, you can instead add the tunneling setup to your `$(HOME)/.ssh/config` file on the host or laptop that you use to login to the cluster:

```
Host clusteralias
    HostName cluster.domain
    User username
    RemoteForward 2222 bitbucket.org:22
```

For more on how to set up bitbucket for SSH, see Atlassians instructions, including how to add an SSH key to your bitbucket profile.

2.1.2 Bitbucket account

To create your own repository, or a fork, or to use the ssh: method behind a firewall, you need to have an account at bitbucket.org.

After registering, use `ssh-keygen` on the cluster frontend to create an SSH key. Then use the "A" icon in the lower left corner of the bitbucket web page to access "Bitbucket Settings" and "SSH keys", and store the public (.pub) part of the key there.

To access a repository or fork of yours from your laptop, or another host, add the corresponding SSH keys in the same way.

2.1.3 Execution environment

To include the `utilities/python/` directory in the Python search path, and (optionally) set default options for compilation, add these lines to your `~/.bashrc` startup file:

```
export DISPATCH=${HOME}/codes/dispatch
export PYTHONPATH=${DISPATCH}/utilities/python:${PYTHONPATH}
# optionally:
export HOST=NameForYourLaptop
```

To set compile options, add a `config/host/NameForYourLaptop/Makefile` containing for example

```
COMPILER=ifort
SEPARATE_BUILDS=on
```

2.2 Experiments

A few of the (many more) experiments from the development repository are made available in the public version, mainly to serve as examples of how to construct experiments, and how to implement solvers. Assuming you have the gfortran compiler and MPI (if not – see *Compiling*), downloading, compiling, and running for example the `heat_diffusion` demo requires only these commands:

```
git clone https://bitbucket.org/aanordlund/dispatch
cd dispatch/experiments/heat_diffusion
make -j
./dispatch.x
```

2.2.1 Heat diffusion

The `heat_diffusion/` experiment demonstrates how to setup an experiment that refers to the very simple solver `solvers/heat_diffusion/solver_mod.f90`, and by extension demonstrates how to add a solver (any solver) to the DISPATCH code framework.

The entire solver and experiment setup is defined by the files:

```
solvers/heat_diffusion/Makefile
solvers/heat_diffusion/solver_mod.f90
experiments/heat_diffusion/Makefile
experiments/heat_diffusion/experiment_mod.f90
experiments/heat_diffusion/input.nml
```

2.2.2 MHD shock

The `mhd_shock/` experiment runs a classical MHD shock tube experiment, where the initial left and right values are set by the input namelist file, and the choice of solver is made by setting the SOLVER macro in the `Makefile`.

2.2.3 Pan

The `pan/` experiment corresponds to the runs behind the results reported in the series of papers by Liubin Pan, Paolo Padoan, and collaborators (see *Publications*, or make a search on ADS for an up-to-date list).

2.2.4 Stellar atmospheres

The `stellar_atmospheres` experiment demonstrates how to setup models of stellar atmospheres, using tabular equations of state, and including diffuse radiative transfer.

2.2.5 Truelove

The `truelove/` experiment corresponds to the runs behind the results reported in the paper

- Ramsey, J. P., Haugbølle, T., & Nordlund, Å., "A simple and efficient solver for self-gravity in the DISPATCH astrophysical simulation framework" , Journal of Physics Conference Series 1031, 012021 (2018). arXiv:1806.10098

2.2.6 Turbulence

The `turbulence/` experiment demonstrates how to set up the driving in forced turbulence experiments, where the details of the driving may need to depend on the value of the SOLVER makefile macro.

The hierachical makefile setup (which may be followed by inspecting the `config/Makefile` and its `sinclude` statements allows one to override the default choice of the `force_mod.f90` or `forces_mod.f90` by placing corresponding files in subdirectories under the `experiments/turbulence/` directory.

The specific driver in the case where `SOLVER=ramses/hydro`, for example, resides in `experiments/turbulence/ramses/hydro/`.

Alternative, as illustrated by the directory `experiments/turbulence/stagger2e/`, one may place a specialized experiment in a subdirectory, where the `TOP` makefile macro is correspondingly defined as `TOP=../../..`.

2.3 Compiling

Typically, you need to use a `module` command to get access to a compiler and MPI library. This could be, for example:

```
module load intel openmpi/intel
```

To compile, go to one of the expriments/ directories, and use `make info` to see macro names and values that would be used in `make`::

```
cd experiments/turbulence
make info
```

If the automatically selected make options shown are OK, just do::

```
make -j
```

If MPI is not available, add `MPI=` to the make command, and if the (gfortran) compiler chosen by default is not appropriate add for example `COMPILER=ifort`). The directory `config/compiler/` shows which compiler configurations are avaialable. See also the *Execution environment* section.

2.3.1 Command line tools

Wether you run on Ĺinux, MacOS, or Windows, you need to have a few of basic command line tools:

```
git
make
gfortran (or ifort)
```

2.3.2 Windows: Cygwin

In Windows, a simple way to get access to gfortan and other tools that are needed (git, make, ..) is to install Cygwin (cygwin.org). Choose the 64-bit version, and be sure to include:

```
gcc-core
gcc-fortran
gcc-debuginfo
openmpi
openmpi-debuginfo
libopenmpi-devel
libopenmpi40
libopenmpifh40
libopenmpiuse08_40
libopenmpiusetkr40
make
git
```

and maybe more (shells, SSH, . . .)

2.3.3 GUI tools

It may also be nice to have GUI (Graphical User Interface) versions of some tools, especially to work with GIT, e.g.:

```
SourceTree
```

2.3.4 Option Groups

Supported compilers come with predefined option groups. To see the actual compiler options they include, do:

```
make OPTS=full_debug info
make OPTS=debug info
make OPTS= info
make OPTS=optimized info
```

The bundled options are chosen so that

- *full_debug*: generates very slow code that traps many problems
- *debug*: generates faster code that traps fewer problems
- *(blank)*: compromise between compile time and code speed
- *optimized*: maximizes speed, at the cost of increased compile time

Note that local options in `config/host/$HOST/Makefile` and in `config/compiler/$HOST/$OPTS.mkf` may modify the option bundles.

2.3.5 Overriding Options

If you prefer to compile with another set of options, do for example:

```
make OPT="-O3 -xHost"
```

Any other of the macro names shown by `make info` can also be replaced:

```
make PAR=
make MPI=
```

would compile without OpenMP and without MPI, respectively.

2.4 Running

To run with the default input file (`input.nml`), with data ouput into the `./data/` directory:

```
./dispatch.x
```

To run with another input file:

```
./dispatch.x run.nml
```

In this case data output goes to the `./data/run/` directory.

2.4.1 OpenMP

OpenMPI is fundamental in DISPATCH, so to take advantage of task parallelization on a single node, set the OMP_NUM_THREADS environemental variable before starting. For example::

```
export OMP_NUM_THREADS=20
./dispatch.x
```

To see how many threads your compute node supports:

```
grep -c processor /proc/cpuinfo
```

That file contains other detailed information about the cores, which might be relevant when choosing compiler options

```
more /proc/cpuinfo
```

2.4.2 Running under MPI

Use the relevant cluster documentation to find out how to run under MPI. To use SLURM to run a job with 64 MPI processes, with two processes per node and 10 cores per process, you may need something similar to this (which would work on HPC.KU.DK):

```
#!/bin/bash
#SBATCH --ntasks=64 --ntasks-per-node=2 --cpus-per-task=10
#SBATCH --mem-per-cpu=3g --exclusive

export OMP_NUM_THREADS=10
./dispatch.x
```

2.4.3 Hyper-threading

If the system supports hyper-threading; e.g., 2 threads per core, with 10 cores per process::

```
#!/bin/bash
#SBATCH --ntasks=64 --ntasks-per-node=2 --cpus-per-task=10 --ntasks_per_core=2
#SBATCH --mem-per-cpu=3g --exclusive

export OMP_NUM_THREADS=20
./dispatch.x
```

2.5 Data Access

To access the data, use Python (cf. ref:*python* and *Execution environment* for setup details). A brief overview of the data structure and a few simple examples of Python access are given below.

2.5.1 Data Structure

Data from execution of experiments ends up under the subdirectory `data/`. It is often convenient to let `data/` be a softlink to a directory on a dedicated disk system (e.g. a Luster file system on a cluster, or an external disk drive on a workstation or laptop). When using the default input file (`input.nml`) data is stored directly in `data/`, while if using e.g. `run.nml`, the data is stored in `data/run/`.

Metadata describing the snapshots is stored as namelists in files such as

```
data/params.nml
data/00000/snapshot.nml
data/00000/rank_00000_patches.nml
```

The binary data may, depending on I/O method, reside in files such as:

```
data/snapshots.dat
data/00000/snapshot.dat
data/00000/00000.dat
```

2.5.2 Simple examples

These example illustrates how to access and display data from snapshot 2 of a run with `input.nml` as input file (see *Execution environment* for the assumed setup):

```
# Open snapshot 2 metadata
import dispatch
sn=dispatch.snapshot(2)
...
# Plot the horizontal average of density
import dispatch.select as dse
z,f=dse.haver(sn,iv='d',dir=2)
import matplotlib.pyplot as plt
plt.plot(z,f)
...
# Display a horizontal plane using YT
import dispatch.yt as dyt
ds=dyt.snapshot(2)
import yt
slc=yt.SlicePlot(ds,fld='density',axis=2,center=[.5.,.5,.5])
slc.show()
...
# Display a horizontal plane using dispatch.graphics
import dispatch.graphics as dgr
dgr.amr_plane(sn,iv='d',z=0.5)
```

For more general cases, see the *Python support* documentation

2.5.3 Data output

Several output formats are avialable, to be described in detail below. In addition, one can optionally add HDF5 output, and “auxiliary” output which is incorporated seemlessly into the Python data access methods.

2.5.4 Auxiliary data

One can add auxiliary data, which will be automatically added to the set of variables accessed via the `dispatch` Python module, by adding these lines in the source code::

```
USE task_mod
USE aux_mod
...
...
call task%aux%register ('name', array)
```

`array` should be a pointer to a 1-D, 2-D, 3-D, or 4-D real array. The `aux` data type may be added to any other sub-class of the `task_t` data type (e.g. as a `patch%aud` or an `extras%aux`.

2.6 Python support

It is highly recommended to use Anaconda as the Python 3 installation, adding also Spyder, and installing the add-on that allows running Jupyter Notebooks inside Spyder.

To import Python modules, add the full path of the DISPATCH directory `utilities/python/` to the Python path (see *Execution environment*), and do:

```
import dispatch
import dispatch.select
import dispatch.graphics
import dispatch.yt
```

2.6.1 Documentation

Several example of how to access and visualize DISPATCH data are given in the *Jupyter Notebooks* section.

The module APIs are self-documented, using the standard Python documentation mechanisms, cf.

```
help(dispatch)
help(dispatch.select)
dispatch.snapshot(<TAB>
```

The source code may be found in `utilities/python/dispatch/`, and is also available via the *auto-generated documentation* (the “Files” menu contains source code).

2.6.2 `dispatch` module

Try for example:

```
import dispatch
help(dispatch)
help(dispatch.snapshot)
sn=dispatch.snapshot(<TAB>
```

To read the 1st snapshot from `data/`, `data/run`, and `../data/run`, do:

```
sn=dispatch.snapshot(1)
sn=dispatch.snapshot(1,run='run')
sn=dispatch.snapshot(1,'run',data='../data')
```

The resulting `sn` is an instance of the snapshot class, where many of the Fortan data type attributes from the source code are available as Python object attributes. To see what is offered, do:

```
dir(sn)
```

2.6.3 `dispatch.select` module

`dispatch.select` contains support functions for choosing selections of snapshot data.

Try for example:

```
s,f=dispatch.select.haver(sn,dir=2)
plot(s,f)
```

which returns in `f` the "horizontal" averages of the density when "up" is axis 2 (the z-axis). `s` contains the coordinate values (z in this case). Or try plotting horizontal min- and max-values, using:

```
s,f0,f1=dispatch.select.hminmax(sn,dir=2)
plot(s,f0); plot(s,f1)
```

2.6.4 `dispatch.graphics` module

`dispatch.graphics` contains support functions for visualizing snapshot data.

Documentation:

```
help(dispatch.graphics)
help(dispatch.graphics.imshow)
```

Note for example `dispatch.graphics.imshow(im)`, which is similar to `matplotlib.pyplot.imshow(im)`, but uses Fortan index conventions.

2.6.5 `dispatch.yt` module

`dispatch.yt` is an interface to the YT Project Python package.

Try for example:

```
import yt
import dispatch.yt
```

To read the 1st snapshot from the `../data/run`, do:

```
help(dispatch.yt.snapshot)
ds=dispatch.yt.snapshot(1,run='run',data='../data')
```

The resulting `ds` is an instance of the YT data set class. For documentation of the various YT commands below, use `help(command)`. To make a slice plot, try:

```
fld='density'
slc=yt.SlicePlot(ds,fld=fld,axis=2,center=[.5.,.5,.5])
slc.set_log(fld,True)
slc.set_colorbar_label(fld,'Density')
slc.set_xlabel('')
slc.set_ylabel('')
slc.annotate_grids(edgecolors='white',draw_ids=False)
slc.show()
slc.save('slice_plot.png')
```

2.6.6 Jupyter Notebooks

Example / template notebooks:

Reading data and 1-D plotting with DISPATCH

NB: Remember to set the `PYTHONPATH` environment variable to `$DISPATCH_DIR/utilities/python`, where `$DISPATCH_DIR` is the location of your DISPATCH repository.

This notebook assumes you have already compiled DISPATCH for the 1-D MHD shock experiment (`make`) and run the code (`./dispatch.x`) successfully. The data can be found in the `data` directory.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import dispatch
import dispatch.select
import itertools
```

First, read a snapshot.

[2]:
```
datadir = '../../../experiments/mhd_shock/data'
snap = dispatch.snapshot(iout=9,verbose=1,data=datadir)
```

```
parsing ../../../experiments/mhd_shock/data/00009/snapshot.nml
  parsing io_nml
    adding property io
  parsing snapshot_nml
  parsing idx_nml
    adding property idx
  parsing units_nml
    adding property units
  parsing cartesian_params
    adding property cartesian
parsing ../../../experiments/mhd_shock/data/00009/rank_00000_patches.nml
  added 3 patches

timer:
snapshot metadata:   0.008 sec
         _patches:   0.003 sec
```

[3]:
```
for p in snap.patches: print(p.id, p.position, p.size, p.time, p.gn, p.ds, p.level)
```

```
1 [0.16666667 0.005      0.005     ] [0.33333333 0.01       0.01      ] 0.09 [66  1 ␣
↪1] [0.00564972 0.01       0.01      ] 7
2 [0.5   0.005 0.005] [0.33333333 0.01       0.01      ] 0.09 [66  1  1] [0.00564972␣
↪0.01       0.01      ] 7
3 [0.83333333 0.005      0.005     ] [0.33333333 0.01       0.01      ] 0.09 [66  1 ␣
↪1] [0.00564972 0.01       0.01      ] 7
```

Printed from left to right are the patch ID, the centre of the patch in Cartesian coordinates, the time of the patch in the current snapshot, and the dimensions of the density/patch.

In this case, although we are dealing with a 1-D MHD shock tube, the solver employed does not permit true 1-D or 2-D calculations and thus there are a few dummy zones in the y- and z-directions.

[4]:
```
indices = snap.patches[0].idx.vars.copy()
print(indices)
print("Patch kind:",snap.patches[0].kind)
```

```
{0: 'd', 4: 'e', 1: 'px', 2: 'py', 3: 'pz', 5: 'bx', 6: 'by', 7: 'bz'}
Patch kind: stagger2_mhd_pat
```

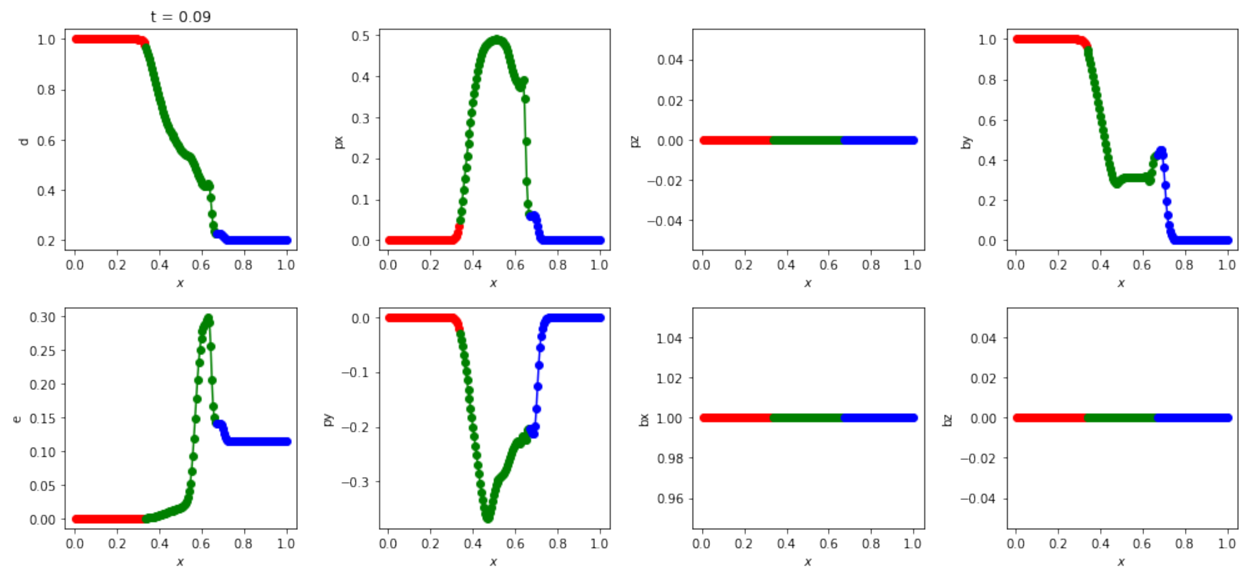
Note: From here on, we assume that the shock tube runs along the *x-direction*.

```
[5]: fig = plt.figure(figsize=(14.0,6.5))
fig.clf()
ncols, nrows = 4, 2
gs = fig.add_gridspec(ncols=ncols,nrows=nrows)
axes = []
for j,i in itertools.product(range(ncols),range(nrows)):
    cax = fig.add_subplot(gs[i,j])
    axes.append(cax)

if 'et' in indices.values(): indices.pop(snap.patches[0].idx.et) # If stored, don't␣
↪plot total energy.

for cax, v in zip(axes, indices):
    colrs = itertools.cycle(['r','g','b','c','m','y','k'])
    for p in snap.patches:
        jslice, kslice = 0, 0
        if p.kind == 'ramses_mhd_patch': jslice, kslice = 4,4
        cax.plot(p.x[p.li[0]:p.ui[0]],p.var(v)[p.li[0]:p.ui[0],jslice,kslice],marker=
↪'o',color=next(colrs),zorder=0.1*p.level)
    cax.set_xlabel(r'$x$')
    cax.set_ylabel(p.idx.vars[v])

axes[0].set_title(r't = {0:.03g}'.format(snap.patches[0].time))
fig.tight_layout()
plt.savefig(snap.patches[0].kind.strip())
plt.show()
```



Here are the MHD variables stored in this patch and their associated index in the data. `px` is the x-component of momentum, etc. You don't have to remember these indices because you can always retrieve them using aliases, e.g., `patch.idx.d`.

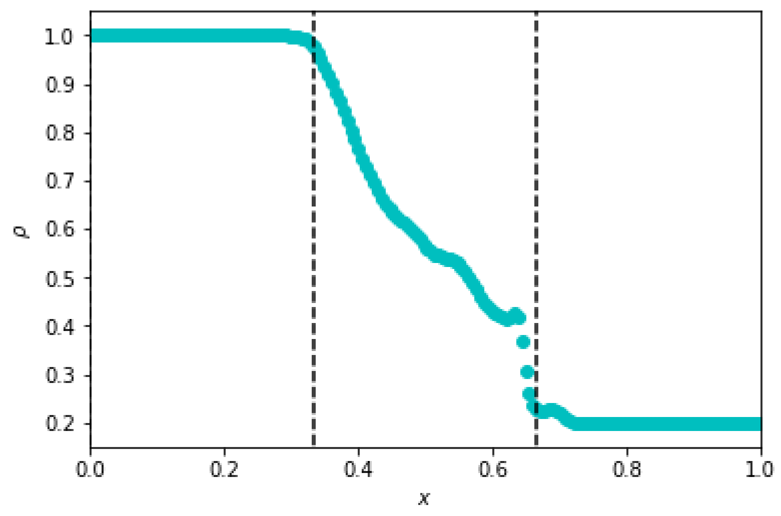
Here's the density at t = 0.09. Different colours have been used for each patch.

Now what if you want to see all of the data as one single array (which can be useful for analysis)?

```
[6]: x, rho = dispatch.select.values_along(snap.patches,[0.0,0.0,0.0],dir=0,iv=snap.
↪patches[0].idx.d)
```

```
[7]: fig2 = plt.figure()
     fig2.clf()
     plt.plot(x,rho,'co')
     for p in snap.patches:
         edge = p.position[0] - 0.5*p.size[0]
         plt.plot([edge,edge],[-10,10],'k--')
     plt.axis(ymin=0.15,ymax=1.05,xmin=0.0,xmax=1.0)
     plt.xlabel(r'$x$')
     plt.ylabel(r'$\rho$')
```

```
[7]: Text(0, 0.5, '$\\rho$')
```



Note that I've manually added vertical lines to denote patch boundaries.

```
[8]: print(rho.shape)
```

```
(180,)
```

As you can see, the data is now available as a single numpy array.

Data access demo

Python packages

```
[1]: import dispatch
     import dispatch.select
     import dispatch.graphics
```

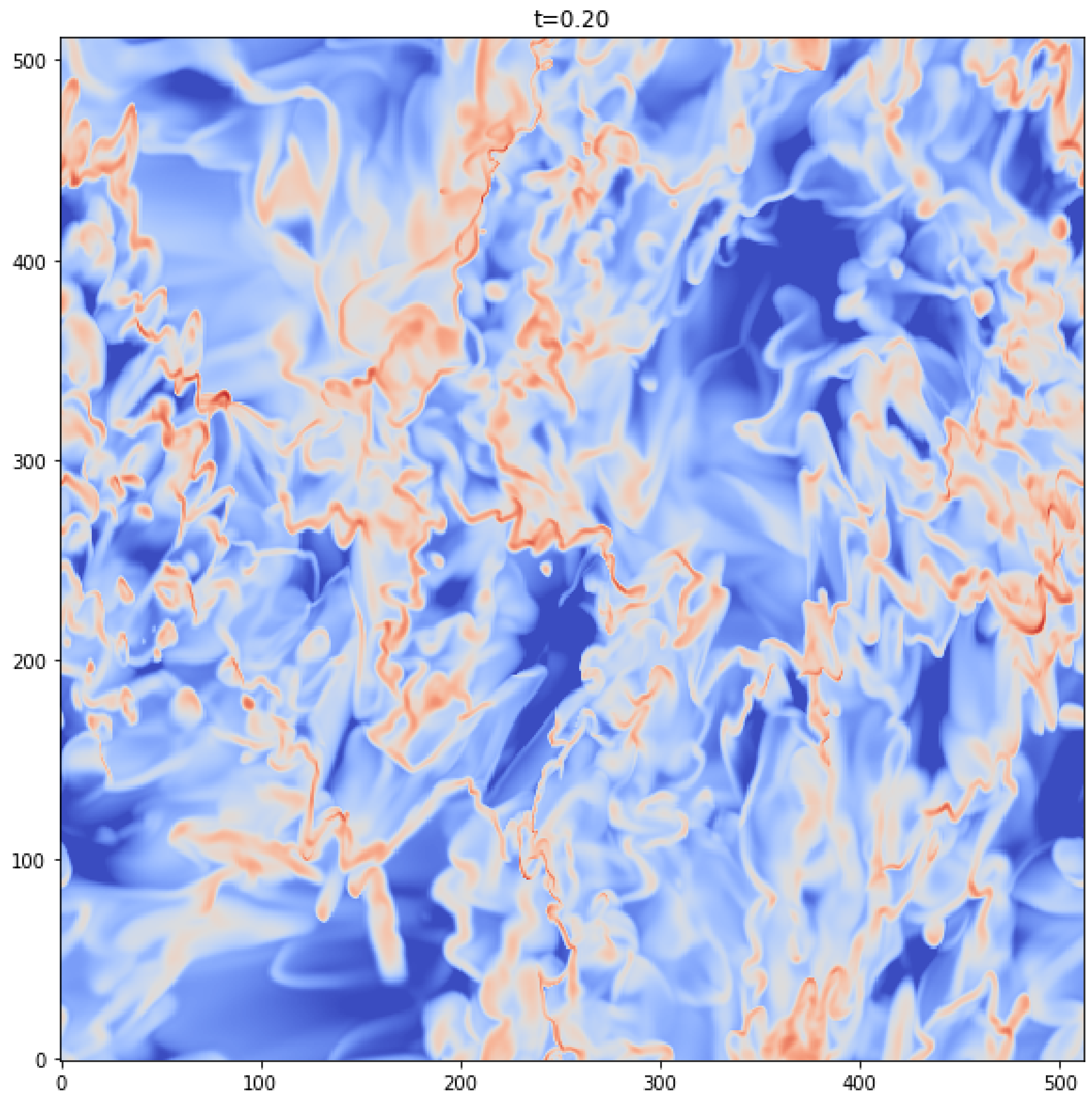
Data and run directories (relative to `../notebooks`)

```
[2]: data='../data'
     run='512m32_hllc'
     io=10
```

Using, from DISPATCH Python packages, `dispatch.snapshot()` to read in metadata, `dispatch.select.uni_gridplane()` to get an array with a slice, and `dispatch.graphics.imshow()` to render it:

[3]:
```
s=dispatch.snapshot(io,run,data)
slc=dispatch.select.unigrid_plane(s)
import matplotlib.pyplot as pl
pl.figure(figsize=(10,10))
dispatch.graphics.imshow(slc,vmin=-5,vmax=5)
pl.title('t={:.2f}'.format(s.time));
```

```
timer:
snapshot metadata:   0.008 sec
         _patches:  11.706 sec
```



AMR slice animation with YT

Module import

[1]:
```
import os
import sys
import time
import yt
import dispatch.yt
import numpy as np
```

Optionally, turn of YT info lines

[2]:
```
yt.funcs.mylog.setLevel(40)
```

Data directory, run directory, and data source field

[3]:
```
data='../data'
run='amr_7'
fld='density'
```

[4]:
```
z=0.6
ds=dispatch.yt.snapshot(30,run,data)
slc=yt.SlicePlot(ds,axis=2,fields=fld,center=[.5,.5,z])
slc.set_log(fld,True)
slc.set_colorbar_label(fld,"Density")
slc.set_xlabel("")
slc.set_ylabel("")
slc.annotate_grids(edgecolors="white")
slc.show()
```

```
<yt.visualization.plot_window.AxisAlignedSlicePlot at 0x2b6ffa7de6a0>
```

Make a sub-directory for images

[5]:
```
os.makedirs(data+'/'+run+'/im',exist_ok=True)
```

Loop over a number of timesteps, producing images for an animation:

[6]:
```
z=0.6                                                          # slice plane
i0=5                                                           # 1st snapshot
i1=50                                                          # last snapshot
im=(i0-5)*2+1                                                  # first frame number
for i in range(i0,i1+1):
    ds=dispatch.yt.snapshot(i,run,'../data')                   # data source
    slc=yt.SlicePlot(ds,axis=2,fields=fld,center=[.5,.5,z])    # slice it
    slc.set_log(fld,True)                                      # use log density
    slc.set_colorbar_label(fld,"Density")                      # replace label on colbar
    slc.set_xlabel(""); slc.set_ylabel("")                     # blank out x- and y-
↪labels
    slc.annotate_grids(edgecolors="white")                     # color of grid annotation
    for j in (0,1):                                            # double frames
        file='{}/{}/im/{:03d}.png'.format(data,run,im)         # .png file
        sys.stdout.write('{:03d}'.format(im))                  # progress
        c='\n' if im%20 == 0 else ' '                          # new line or not
        sys.stdout.write(c)
        im+=1
        slc.save(file)
```

```
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020
021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040
041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059 060
```

(continues on next page)

(continued from previous page)

```
061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079 080
081 082 083 084 085 086 087 088 089 090 091 092
```

2.7 Setting up experiments

To construct a new experiment (*experiment* is essentially synonymous to *simulation* or *project*) in DISPATCH one can often start out from an existing experiment, clone it into a new `experiments/whatever/` directory, and modify or replace the existing files. In fact, an experiment is *completely* defined by the (relatively few) files that are present in the specific `experiments/whatever/` directory.

The main files present there, and their funcionalities, are:

1. `Makefile`: select the main solver responsible for evolving the experiment
2. `experiment_mod.f90`: specify initial conditions (ICs) and boundary conditions (BCs)
3. `scaling_mod.f90`: specify scaling between physical and code units
4. `extras_mod.f90`: select additional functionalities from the `extras/` directory, such as *selfgravity*, *radiative transfer*, *auxiliary HDF5 output*, etc.
5. `run.nml`: specify, in Fortran namlist input files (`*.nml`) the parameters of specific *runs* of the experiment.
6. `python/*.{py,ipynb}`: provide Python support; e.g. for preparation of input and/or analysis of output

2.7.1 Makefile

The `Makefile` should initially be copied from another experiment directory, and usually requires only minor editing, of mostly obvious nature – such as choosing a different solver, changing the default compilation options, and adding or modifying the `extras_mod.o:` dependencies.

2.7.2 experiment_mod.f90

The `experiment_mod.f90` module defines the highest level data type, which *extends* (inherits) the `solver_t` data type, and optionally overloads (or “intercepts”) calls to the standar `init`, `update`, `output`, and possible other procedures. The main purposes of these are:

1. `init`: read in and use input parameters from an input namelist, by convention called `experiment_params`.
2. `update`: do whatever specific actions that are required (or not) before and after calling the solver update procedure to update the state of a single *patch* (or more generally: *task*)
3. `output:` this is called immediately *before* call to the `update` procedure (because that’s the point where ghost zones have been loaded, boundary conditions have been applied, etc).

Any or all of these procedures may be omitted, in which case calls are instead caught by the corresponding `solver_t` procedures. One can also pass on calls to these, with for example:

```
SUBROUTINE output (self)
  class(experiment_t):: self
  !.......................................................................
  call self%solver_t%output
  ... whatever else one may want to do, e.g. via the HDF5 interface
END SUBROUTINE output
```

2.7.3 scaling_mod.f90

The `scaling_mod.f90` specifies the relations between physical and code units. To create it, copy the template file `microphysics/scaling_mod.f90` (or copy an existing `experiments/*/scaling_mod.f90` file).

The contents should be essentially self-explanatory. By choosing three scaling units, for example *length*, *time*, and *mass*, all other units are defined.

The three basic units may be expressed in CGS or SI units; both units systems are present in the template file.

2.7.4 extras_mod.f90 options

The `extras/` directory, and the related `extras_mod.f90` template file provide optional extra features, which are only linked into the executable if explicitly selected.

In practice, this is done by copying the `extras/extras_mod.f90` template file to the `experiments/whatever/` directory, uncommenting lines related to the selected extra features, and adding a line listing those features in `experiments/whatever/Makefile`.

2.7.5 *.nml input files

`input.nml` is a template input file, which serves a dual purpose:

1. Define a simple test case, that should not take more than about a minute to run on an average laptop
2. Act as a template file, to be copied to input files `whatever.nml` whose names also define the location of output files, which appear in `data/whatever/` directories. Here `whatever` may be chosen arbitrarily, and could for example be `run1`, `run2`, or `2019-08-22a`, or `L=30px_levels=6-9`.

2.7.6 python/*.{py,ipynb} files

Use a `python/` sub-directory to store `*.py` and `*.ipynb` (Jupyter Notebooks). This is where to put the *experiment specific* Python files.

Functionalities that may be used in any context should instead be added to `utilities/python/`, or should be amended to for example `utilities/python/dispatch/_dispatch.py` or other files there.

For details about data access via Python, see the *Data Access* section.

CHAPTER 3

Technical Details

Note 1: some of the issues and features described here may for a period of time only be available in the development repository.

Note 2: some of the technical discussion may not be up-to-date – they are left visible to encourage updates.

3.1 Adaptive Mesh Refinement

3.1.1 Refine event sequence

- In `refine_t%make_child_patch()`
  - create the new patch
  - add its parent patch as the only nbor, and prolong
  - set `bits%init_nbors`
  - add it to the task list
  - add it to the ready queue
  - switch perspective to the first time it enters `refine_t%check_current()`
- In `refine_t%check_current()`
  - detect `bits%init_nbors`
  - generate a new nbor list
  - possibly set `bits%init_nbors` on the nbors
  - clear `bits%init_nbors`
  - continue the usual business
- In `task_mesg_t%unpack()` for virtual tasks
  - must do whatever an active task does; using `list_t%add_new_link` for that

  - must also detect `bits%init_nbors`, and act accordingly

3.1.2 Derefine event sequence

- Call `list_t%remove_remove_and_reset()` to
  - set bits%remove, which hides the task in check_ready() (task and nbor)
  - remove the task from the task list (it will remain in garbage bin while needed)
  - use the existing nbor list to set `bits%init_nbors` on nbors
  - now that the task is removed, call `check_nbors()` to see if nbors have become ready
  - move the task to the garbage bin
  - when the task is no longer needed (no longer is a member of an nbor list) the task is deallocated and deleted by `list_t%garbage_remove()`
- In `task_mesg_t%unpack()`
  - detect `bits%remove` and call `list_t%remove_remove_and_reset()`

This is identical to what is done on the owner rank, and is taken care of by the `list_t%remove_and_reset()` procedure.

3.1.3 New AMR tasks

A new AMR task (A) is ready to update immediately, since it was spawned from a task that was ready to update at that specific time. To update, with guard cells filled, it needs to have an nbor list. The nbor list must have the required nbors, and the nbor links must have `needed` and `download` both set to true.

When that task (A) nbor list was created, and indeed when any new nbor list is created, that very action should trigger an `init_nbors()` in the nbors of the A task (but not in nbors of nbors!). So, the caller of the `init_nbors()` must also call `set_init_nbors()`, which set `bits%init_nbors` in all nbors of the 1st task.

One of those nbors (B) may be virtual, and that tasks should also get new nbor lists, both on the rank (a) that triggers the chain of events, and on the rank (b) that are owns task (B).

The rank (b) that owns the virtual task (B) also has (or will get) a copy of the task (A) that triggered the event chain, and since the principle is that the virtual tasks behave exactly as their originals do, we need only to ensure that the thread on rank (b) that unpacks the virtual copy of (A) also performs a `set_init_nbors()` call, which then triggers an `init_nbors()` in the next update of the boundary task (B) on rank (b), which then gets send in copy to rank (a), as un update of its virtual task (B), where it gets a new nbor list, *provided* that the `bits%init_nbors` that caused the thread on rank (b) to generate a new nbor list still is set when the copy of the task arrives in rank (a).

Hence, when a thread updating a boundary task discovers a `bits%init_nbors`, it should not clear that bit until after the `send_to_vnbors()` call.

3.1.4 Adding a new task

When a thread adds a new task, either as a consequence of AMR on the same rank, or because of AMR on another rank, or because of load balancing, it must always take these actions:

- Make sure that `class(patch_t)` tasks have a link back to the task link
- Add an nbor list to the task link

- Cause the new task to be added to the nbor lists of the nbors, by calling `list_t%set_init_nbors()`, which sets `bits%init_nbors` in all nbor tasks.
- Increment the task total and task level counts on the rank
- Call `check_nbors()` on the new task link, which then (as always) runs `check_ready()` on all the nbors first, and then runs check_ready() on the task link itself (unless it is a virtual task).

These actions are taken in a procedure `list_t%add_new_task()` that is called from both the AMR procedure that created new tasks, and from the `task_mesg_t%unpack()` procedure that creates virtual copies of new tasks, or that creates a new virtual task because that task became a boundary task on the rank that owns it.

3.1.5 Removing a task

When a thread removes a task, either as a consequence of AMR on the same rank, or because of AMR on another rank, or because of load balancing, it must always take these actions:

- Immediately set `bits%move` bit in the task status (under a brief lock). That bit should make the task effectively removed, immediately. Hence, even if it still exists, and is in the nbor lists of some tasks, it is ignored in `check_ready()` calls.
- While the task is being processed for removal, it is placed on a garbage list, as it may still be needed in task updates. While that is going on, the task may still be used in `dnload()` procedures.
- When the access count (n_needed) reaches zero, because the task has been removed from all nbor lists where it was present, the garbage collection routine is free to actually deallocate everything, and remove the last traces of the task.

To be precise on a particular aspect here: As long as a task that used to have it in its nbor list has it there, it needs to be able to provide guard cell data. But when the nbor list has been update and the task in question is not there anymore, then the task list presumably contains other tasks that provide the same guard cell coverage, and hence the task may be actually removed. Hence; the use in `dnload()` is tied one-to-one on the presence in the nbor list, so `download_link()` should not take the remove bit into account.

The functionaly described above is taken care of by `list_t%remove_and_reset()`

3.1.6 Garbage collection

When tasks are being added or removed it is convenient if threads the are still assuming that the tasks exist can finish their work, without errors or task locking. This can be achieved by maintaining a count `task_t%n_needed`, which keeps track of how many threads that currently have a lock at an nbor list where the task is a member.

If the number is larger than zero, a task that should be deleted is instead added to a list of garbage task, from which it is removed and deleted when the count reaches zero.

There are only three procedure that create or delete nbor lists: `copy_nbor_list`, `sort_nbors_by_level`, and `remove_nbor_list`. These maintain the `n_neeeded` count, using atomic increments and decrements.

3.1.7 Nbor list dead-lock

A deadlock could occur if a task A depends on another task B being updated, because B is in the nbor list of A and the logical flag `needed` is true for the B entry in A's nbor list, while at the same time A is not in the nbor list of B, and hence B will not include A when doing `check_nbors()`. This could cause a deadlock because the mechanism that puts task A in the ready queue is that a thread working on B, updating the time past the one that task A is waiting for, then calls `check_ready()` on the A-task.

But if the reason that the A task is not on the nbor list of B is just a time delay (e.g. because of latency in MPI communication), then one just neeeds to make sure that the `check_ready()` call actually happens when A – after the delay – gets put on the nbor list of B. This will be ensured if the adding of a task to an nbor list always is accompanied by a `check_ready()` call. One call too many does not hurt.

So, at guarantee against nbor-list caused deadlocking is to add a `check_nbors()` call into the `init_nbors()` call. This will work, as long as `init_nbors()` is the method used to update the nbor lists from `task_mesg_t%unpack()`. If the method is changed the new method needs to do something similar.

3.1.8 Nbor list handling

If threads were to be allowed to update the nbor lists of tasks that another thread might be working on, the situation would become complicated. The thread would need to lock the nbor list of the other task while updating it, but it cannot be allowed to lock it's own nbor list during that time, since the thread working on that task might happen to be running the same procedure at the same time, and a deadlock could then occur.

If threads only ever change the nbor list of their active task, and leave changing the nbor lists of other tasks to the threads updating those tasks, then much of the nbor list locking and copying can be avoided.

When the active task accesses the nbor list of another task – this happens primarily in `check_nbors()`, it needs to be sure that the another thread doesn't change the nbor list in the mean time, so it needs to lock it. That locking is only effective if a thread that updates its own nbor list actually locks it when it does so. Hence even with read-only access, the task owning thread must lock the nbor list link while changing the nbor list.

Currently the generation of new nbor lists is primarily done by `list_t%init_nbors()`. If/when other procedures are used to modify nbor lists they must lock the task link while doing so.

3.1.9 Nbor list protocol

The nbor list handling is based on these principles:

1) Threads need to lock their nbor lists while updating them. This has the extra benefit that the updating does not require copying the nbor list in the mean time, except for the benefit of making the lock time as short as possible. However, considering how relatively seldom nbor lists need to be updated, locking during update is the simplest and safest.
2) While accessing the nbor lists of other tasks (e.g. during `check_nbors()`), threads must acquire a lock on the nbor list while using it, to ensure it isn't being changed in the meantime. However, it must not modify the nbor lists of other tasks
3) The nbor lists presented to other tasks need not be completely up-to-date, but they must be *valid*, in the sense that the nbors exist and have the expected time advance.
4) If at some point in (wall clock) time a task (A) has an nbor list that contains an nbor (B) that does *not* have task A in its nbor list, then to avoid deadlocks, a `check_ready()` on task A must happen as a consequence of it being added to the nbor list of task B.

The lines with task memory locking should be marked with "TASK LOCK", while the nbor list locking should be marked with "LINK LOCK":

3.1.10 Nbor task access protocol

In addition to accessing nbor tasks nbor lists, the thread updating a task also needs to access and modify nbor task structures. Setting bits in the status word is inherently (by construction) atomic, so that should cause no problem, and should not require task locking. Changes of `patch_t%mem` should be protected by setting `task_t%lock`. This includes

- `task_t%rotate()`, where the memory slots are rotated, to create a circular buffer
- `timestep_t%update()` for `stagger2/` solvers, and corresponding sections of other solvers, where their `patch_t%mem` arrays are being modified

The direct costs of applying such locks by the owner thread are negligible. Locking other tasks while accessing their memory could impact a more significant cost, and should be avoided when possible.

A relevant example is the access of `patch_t%mem` from other tasks in the form of `source%mem` in `download_mod.f90`. In principle, one should apply a lock there, but the chance that something bad happens is very small, since most of the time, the access will be to two pairs of memory slots that have essentially zero chance of being modified during the access. Since the access time in `download_mod` is a tiny fraction (of order 0.1%) of a task update time, if the source task happens to be under update (the chance is a few %, since most task are dormant at any one time), the update is not likely to finish during the brief access time. Therefore, in this case it is necessary to lock the source task during access only when the oldest slot that is accessed corrsponds to `task_t%new` in the source.

3.1.11 Handling `bits%init_nbors`

When a thread updating a boundary task discovers a `bits%init_nbors`, it should not clear that bit until *after* the `send_to_vnbors()` call, since the bit needs to be propagated to virtual copies of the task.

The nbor list should not be updated too early, since the thread is updating the task because it was deemed ready to update based on the existing nbor list, which has then been used (before the call to `task%update()`) to load data into the guard zones. The existing nbor list might also be used for some unspecified reason as part of the update, so it should not be touched until after the task has updated.

Hence the `init_nbors()` call should be done by `task_list_t%update()`, *after* its call to `task%update()`, and before the call to `send_to_vnbors()`.

The call to `send_to_vnbors()` is immediately preceeded by a call to `load_balance()`, which means the boundary task that will have a new nbor list generated could possibly be reassigned to another rank. It will, in any case, since it arrives with `bits%init_nbors` set, have a new nbor list generated also on the other rank, so the result will in any case be that both the first rank and the other rank will have consistent nbor lists for the task.

The state of that task after the `task%update()` and `init_nbors()` is that of a "dormant" task, ready to be checked as a candidate for the ready queue by any thread updating one of its (possibly new) nbors. That is at it should be.

3.1.12 Consistency with MPI

The removal or addition of a task that is a boundary task should be reflected in the nbor list of nbors that are virtual tasks, and there could possibly be a signficant delay before that happens. The sequence of events should be:

- A thread always sends a boundary task to rank nbors, after updating or creating it. At that time, a bit (`bits%init_nbors`) must be set, which triggers the receiving rank to update the nbor lists of the task; either creating it if is a new task, or removing it and the task, if the task is to be removed.
  - So, when should an `init_nbors()` call be made from `task_mesg_t%unpack`? Clearly, when a new task is being created, but also when a task that is an nbor is removed or one that should become one is created. The first type causes a call automatically (should it?), while the 2nd type is triggered by the `bits%init_nbors` being set.
  - The first type occurs when a task has just been created by AMR, and for a boundary task it happens both on the owner rank and on virtual nbor ranks.
- As a consequnce of that, all nbors of that (virtual) task on the rank nbor should also have their nbor lists renewed. This includes those boundary tasks that appear as virtual tasks on the first rank.

- As the nbor lists of those virtual tasks are being modified, they thread that is doing the modifications must, correspondinly, take the appropriate action (i.e. calling check_nbors() for new tasks, as well as for tasks that are being removed.

3.1.13 MPI action items

Action items to ensure correct handling of AMR task under MPI:

- Ensure that new AMR tasks added are created with a valid nbor list = one that may be used to fill its guard zones in the first update. [x]
- Ensure that new AMR tasks are immediately added to the ready queue, since there is no other mechanism to put them there, and since they are indeed ready to be updated. [x]
- Ensure proper handling of `bits%init_nbors`, which should trigger a call to `init_nbors()` from `task_list_t%update()`, after its call to `task%update()`, and before its call to `load_balance` for active tasks, and a similar call to `init_nbors()` from the `task_mesh_t%unpack()` procedure for virtual tasks. [x]
- Ensure that the nbors of a new task, as well as of task to be removed, get their `bits%init_nbors` set, and that that bit travels with boundary tasks as is acted upon by the `task_mesg_t%unpack()` procedure when it unpacks virtual tasks. [x]
- Ensure that any `add_nbor_xxx()` call is accompanied by a `check_ready()` call of that task link. This is done by adding a `check_nbors()` to the end of the `init_nbors()` call that possible added a new task. [x]
- Ensure that no locking of other task links occur. The choice is made by not setting `omp_lock%links` in task%refine, and to make this choice permanent, all section of the code with `if (omp_lock%links) ...` should be removed. [x]
- In addition one should search explicitly for `%lock%set` in the code. [ ]

3.1.14 MPI I/O for AMR

Currently, only the `method='legacy'` file format works with AMR, and then only for writing – restarts have not been implemented yet. A new `method='snapshot'` will be implemented for efficient saving of AMR data in a single `data/run/NNNNN/snapshot.dat` file per snapshot.

The structure of data on disk should be optimized for fast reading, and should have a format that is independent of the MPI geometry. This is achieved by (temporarily) numbering all patches in a rank `1...np`, where `np` is the number of patches at the time of writing. Then such blocks from all ranks are arranged with each variable filling a contiguous piece in the snapshot.dat file; viz:

```
r1p1...r1pN, r2P1...r2pN, . . .,rMp1...rMpN
```

where `M` is the number of ranks and `N` is the number of patches per ranks (which in general is allowed to differ from patch to patch). A metadata file gives, for each sequence number, the offset into the file, and the previous task number::

```
seq  rank var task offset
 1    0    0   1    xx
 .
 N    0    0
 .    .
 N    M    0
 .    .    .
 N    M    V
```

A restart does not need to result in the same tasks residing on the same rank as before, but should achieve that whenever possible. After or while reading in the metadata, each rank reads in as many as its share of the load

3.1.15 Snapshot save

A saved AMR snapshot must contain – effectively – a list of tasks to read in, initialize, and fill with values from the snapshot.

In the case of for example the `ppd_zoom` experiment, the task list consists of more than one piece, with different types of task in each piece. Sufficient info about this must be carried by the snapshot meta-data.

In general, a "component" that is part of the simulation setup ("scene") may need to have its own save/restore (output/input). If so, it also means that patches and tasks need to be identified as belonging to a component, either by being part of a partial task list, or by having an identifier string (or integer). The former is not convenient, since on the one hand we want to have all tasks in a rank-global task list, and on the other hand we need to be able to add and remove tasks in only one place, not in the global and partial lists separately.

One could possibly rely on that tasks needing special I/O treatment probably also need extra, non-generic parameters, and hence need to belong to a certain type of task, which then would naturally have its own input/output procedure.

3.1.16 Snapshot restore

When reading in AMR tasks, one needs to know the task type to be able to create the task, into which the data is to be read. The sequence of steps needs to be

1. read the meta-data of a task
2. create the task
3. load data into the task

Clearly, the meta-data needs to contain information both about the task type, and about the location of the task data on disk.

It would be nice if this could be semi-automatic, so detection of task type was done by a required data type method.

A part of this issue is also the arrangement of different task types. We should be able to allow different solvers to exist, without insisting that one must be the extension of another. They should all be extensions of task_t, and some should all be extensions of patch_t::

```
          task_t
          /  |  \
         /   |   \
  task1_t    |    task2_t
             |
          patch_t
         /   |   \
        /    |    \
patch1_t     |     patch2_t
```

3.1.17 PDF I/O for AMR

The `pdf_io_mod.f90` module outputs snapshots of the density *probability distribution function* (PDF) at regular intevals (`out_time``in the namelist ``pdf_io_params`). The PDF is accumulated as patches progress past the next output time (`patch_t%pdf_next`), at which time the density data is interpolated to the exact output time its contribution is accumulated in `pdf_io_t%counts`.

When a new AMR task is created, `pdf_next` is set to the next multiple of `out_time`, thus triggering a call to `pdf_io_t%update` when the task reaches that time. The number of tasks that need to be accumulated before the PDF is complete is (provisionally – cf. below) equal to `io%nwrite`, which is also the number of tasks that contribute to AMR snapshots of the patch values.

When one snapshot of the PDF finishes, the count of the number of tasks expected in the next snapshot is set equal to the number of tasks existing at that time, which is the value of `io%nwrite` at that time. If/when new AMR tasks are added, with `pdf_next` set to the next multiple of `out_time`, this increases the number of tasks that should be output with one. The counter that keeps track of tasks remaining to be output should thus be incremented by one. Correspondingly, if an AMR task is deleted, the count of remaining tasks to accumulate should be decremented by one.

There may be borderline cases, where a task is created just after the time when an output should happen, but these should really not caue problems, if the procedure above is followed.

3.1.18 Level support

The AMR actions on a task – refine and derefine – should only be done by the thread that is about to update the task, since any other alternative would lead to serious synchronization problems. The new level `L-1` tasks that may be needed to support a new level `L` task can thus not be created until the next time the level `L-2` task that needs to be refined is updated.

The `refine_t` data type has two methods related to checking for AMR support (i.e., checking that all patches at level `L` have level `L-1` nbors to get guard zone values from):

1. `check_support()` checks if a new (child) patch actually has support. If not, it issues a WARNING in the rank log file. The task then takes guard zones values from `L-2` nbors, until the relevant `L-1` nbors have been created.
2. `need_to_support()` checks if a given `L-2` level patch needs to be refined, to provide support for existing level L patches. If so, it creates such child patches(), along with any other refinement that may be detected a need for in `refine_t%needed`.

Both methods use 2-byte integer maps to detect lack of support – this method will work unmodified for moving patches.

A status bit (`bits%support`) is available, and is used to communicate to nbor patches about the need for support.

A new child patch is automatically sent vbor ranks when it is created, and virtual patches are created there, as when any task arrives, with and id that doesn't exist yet on the rank. The `bits%support` informs the rank that this is a new AMR child patch.

Any new task arriving to a rank generates a call to `list_t%init_nbor_nbors()`, which uses position information to first create an nbor list for the task itself, and then generates new nbor lists also for the nbors of the task, since both sides of the nbor-nbor relation need to have consistent nbor lists.

3.1.19 Flux consistency

For a property such as mass to be conserved across patch boundaries, it is necessary that the two tasks agree, at least on average, on the mass flux passing through the boundary between the tasks. In the guard zones, task A can only estimate the flux computed and used by task B, especially if they have different resolutions and/or timesteps, since the flux for example depends on slope limiters that act differently for different resolution, and since flux calculations are based on forward predictor steps that result in different values at different times.

The solution is that tasks communicate the flux that was used in previous timesteps, so nbor tasks can compensate – *a posteriori* – for differences. The flux must be defined as exactly the flux that was used in the update. To keep and communicate it, one needs an array size of 6*N*N per time slice and per variable.

Since MPI bandwidth is usually not a bottleneck, one may – as a simple temporary solution – increase `mw` by one and communicate the flux values for the whole patch. Optimization is straightforward and can be done later – the main goal is to demonstrate that flux consistency can be achieved, and how it should be done.

To explore this, the sub-sections below progress from the simplest case to the most general case:

Same resolution and timestep

Assume two task with the same resolution and the same timestep size are being updated by one thread::

```
0------------+-----------------+----------------+------------+--- task A
|----fA1-----|------fB2--------|
|----fA1-----|------fB2--------|
0------------+-----------------+----------------+------------+--- task B
```

Assume the thread updates task A first, using an estimate of the flux through the interface between them computed based on internal values from A and values in the guard cells obtained from task B. The flux is saved, and is made available to task B.

Then the thread updates task B, and would in fact compute exactly the same flux at the interface, since it would have the values from task A in its guard cells. Hence it doesn't matter which of the fluxes it uses, and there is no need for flux corrections.

The same holds true if the two tasks are updated simultaneously by two threads.

Same resolution, different timesteps

Assume two task with the same resolution and different timesteps are being updated by *one* thread::

```
tA1              tA2               tA3
 |<---dtA1------->|<----dtA2------->|
 0----------------+-----------------+----------------+------------+--- task A
 |       ´    |   |           |     |
 |-----fA1----|fA1|----fB2----|-fA2-|
 |            |   |           |     |
 0------------+---------------+--------------+-----------+------------ task B
 |<--dtB1---->|<----dtB2----->|<---dtB3----->|
tB1          tB2             tB3
```

Assume the thread updates task A first, and that its timestep is a bit longer than that of task B. It makes the flux it used (fA1) available to task B.

Then the thread updates task B, and computes a somewhat different flux than task A (because the flux is based on prediction of the state at a different time). If it uses that flux, the mass flux between the two tasks becomes inconsistent, and mass is either gained or lost.

To use consistent fluxes, without changing the state of task A a posteriori (and there is no reason to do that – it made an accurate estimate), task B should use the flux from task A (fA1) in the 1st time step.

The thread will then select task B again, for a 2nd update, because task A is ahead of task B. Here, task B should use the remaining part of the task A flux, and then use the flux (fB2) it computed (possibly even correctly time centered), for the remaining part of the time step.

The next task to be updates is task A, which should use flux fB2 from task B for the first part of the interval, and a flux (fA2) that it estimates itself for the last part of the timestep.

Correctly centering the partial timestep fluxes would mean to use predictor values at, for example `tB3 + (dtA2-(tB3-tA2))/2` instead of at `tA2 + dtA2/2`.

Same resolution, time extrapolation

Assume two task with the same resolution and the different timesteps are being update by *several* threads, with a `grace` value giving an allowed window for time extrapolation::

```
tA1              tA2               tA3
 |<---dtA1------->|<----dtA2------->|
 0----------------+-----------------+----------------+------------+--- task A
 |-----fA1--------|------fA2--------|-------fA3------|
 |-----fA1----|fA1|----fB2----|-fA2-|---fB3--|
 0------------+---------------+--------------+------------+----------- task B
 |<--dtB1---->|<----dtB2----->|<---dtB3----->|
tB1          tB2             tB3
```

Assume that after task A and task B have updated as in the previous case, one thread takes task B to do the 2nd time step of it, while another one takes task A, which has also been deemed ready to update, because it is within the grace interval ahead of task B.

Task B will update as in the previous example, using first the fA1 flux from task A, and then its own fl2 flux. Task A will use its own flux (fA2), since task B has not yet made fB2 available. This creates and inconsistency, which needs to be corrected subsequently.

To avoid imposing any new constraints on which task updates before the other, we assume that task A is still ahead by less than the `grace` value, and hence the two tasks could happen to be updated simultaneously again.

The inconsistency in the fB2 interval may be rectified if task A adds (fB2-fA2) times the length of the time interval. Task B is free to choose to use fA2 in the overlapping time interval, and then its own fB3 value, while task A cannot use new information from task B regarding fluxes, since none is available, and hence it will use fA3 over the whole interval. The situation after the update is thus the same as after the previous update, and may be corrected in the same way, in the next round of updates.

Different resolution

If the two task differ in resolution, with a constant ratio, and has aligned cells, then the only difference is that the flux contributions need to be normalized by the cell area, and needs to be summed over the same areas.

Maintaining consistency

To maintain consistency each task must

1. Maintain a time for each interface, up to which it has achieved flux consistency.
2. Have access to the corresponding information from the task on the other side of the interface.

Each task must be able to rely on that both tasks make the same assumptions about update order and method to resolve flux inconsistency. To achieve this without a need for negotiations, we can use the simple rules that 1) the "consistency marker" must always move forward in time, and 2) the task that lags behind in consistency time has "the right-of-way"; it can move its marker ahead of the other task.

Task B cannot know (should not need to know) if task A will be updated during its own update (it could check the `bits%busy` bit, but that might be set only after the task B update has started).

If the consistency time of task A is ahead of that of task B then, by the first rule above, both task A and task B can trust that fluxes agree up to the earlier of the two consistency times. By the 2nd rule, the leading task can trust that the other task respects its flux choice(s) in the time interval between the two consistency times.

So, the rules holding at an interface are:

1. the task which leads in consistency time can assume that the other task will adjust the fluxes used, up to the leading time
2. the task which lags in consistency time has the obligation to fix consistency up to the leading time, and should then move its own consistency time on ahead to the end of its time step

```
|<---dtA1------>|<----dtA2------>|
0---------------+----------------+----------------+------------+--- task A
|-----fA1-------|------fA2-------|-------fA3------|
|-----fA1----|fA1|----fB2----|-fA2-|---fB3--|
0------------+---------------+--------------+------------+---------- task B
|<--dtB1---->|<----dtB2----->|<---dtB3----->|
```

Conserving diV(B)

The divergence of the magnetic field should remain equal to zero at all times and at all locations. This condition is a *constraint* on the magnetic field components passing through a cell, from which one can determine the flux through one face from the fluxes through the other five faces. This may be used to enforce div(B)=0 at the interface between two DISPATCH patches, with components centered like so:

```
     |        ||        |
     Bx  Bz   Bx   Bz   Bx
     |        ||        |
=====+===By===++===By===+=======
     |        ||        |
     Bx  Bz   Bx   Bz   Bx
     |        ||        |
-----+---By---++---By---+-----
     |        ||        |
     Bx  Bz   Bx   Bz   Bx
     |        ||        |
-----+---By---++---By---+-----
     |        ||        |
     Bx  Bz   Bx   Bz   Bx
     |        ||        |
=====+===By===++===By===+=======
     |        ||        |
     Bx  Bz   Bx   Bz   Bx
     |        ||        |
```

The double line symbolizes the patch boundary, with the Bx magnetic field component defined at the same location by two different tasks, generally at a sequence of different times

The continuity of By(x) and Bz(x) through the face is guaranteed, since any glitch would correspond to an electric current, which would generate a counter-acting force.

The continuity of Bx(x) may then be enforced by simply computing the Bx component at the interface from the other 5 known face flux values: Bx(face) = Bx(internal) - delta(By) - delta(Bz) (where the delta() has to be taken in the appropriate sense).

Initially, the values of By(z) at the upper y-edge, and the Bz(y) values at the upper z-edge are not known, but they are subsequently constructed, so after a few iterations also the edge and corner values are consistent with div(B)=0, and the whole cube, including the ghost zones around it, is divergence free (if it was to begin with).

3.1.20 Child task management

Steps when a new AMR task is created

1. The task is created, with no nbor list, and w/o being present in any nbor list, with only interior interpolated data from the parent
2. It is sent directly to the ready queue
3. The thread call check_current, notices a bits%init_nbor, and generates a new nbor list with `tlist%init_nbor (link)`
4. That is enough to download guard cell data
    - Some cells may need to be interpolated from L-2 nbors, which must be there, since otherwise the parent patch would not have had support
    - This is quite OK, since the data are interpolated in any case
5. The procedure also sets bits%init_nbor in the nbor tasks, but does not itself generate new nbor lists for its nbors
6. When the nbor tasks come up for updating, they too will generate new nbor lists, because of the bits%nbor status bit
7. With the nbor lists initialized, everything has arrived at a new order, and the next `check_nbors()` call will include testing of the new child patch.

Using this strategy, we are able to avoid calling `init_nbor_nbors()` after creating a new AMR task, and we are also able to avoid calling `check_nbor_nbors()`, since the nbors will take care of their own `init_nbors()`.

3.1.21 Event sequences

If threads only ever change the nbor list of their active task, and leave changing the nbor lists of other tasks to the threads updating those tasks, then much of the nbor list locking and copying can be avoided. The choice is made by not setting `omp_lock%links` in task%refine

When the active task accessed the nbor list of another task – this happens primarily in `check_nbors()`, it needs to be sure that the another thread doesn't change the nbor list in the mean time, so it needs to lock it. However, it does not need to lock it's own nbor list during that time.

Each thread needs to lock its own active tasks nbor list in `init_nbors()`, while it is being changed, and then needs to lock the nbor's nbor lists when they are being used (`in check_nbors()`).

Each thread needs to lock it's own task memory while it is being changed (inside the hd_t%update or in timestep_t%update), and then needs to lock the memory of nbors as it accesses it (in `download_t%download_link()`).

3.1.22 nbor list consistency

The nbor lists of two tasks must always be *consistent*, in the sense that no deadlock can occur because of the state of the nbor lists of two task that may or may not be nbors.

A deadlock could occur if a task A depends on another task B being updated, because B is in the nbor list of A and is the logical flag `needed` is true for the B entry in A's nbor list, while at the same time A is not in the nbor list of B, and hence B will not include A when doing check_nbors(). This could cause a deadlock because the mechanism that puts task A in the ready queue is that a thread working on B, updating the time past the one that task A is waiting for, is supposed to be that the thread updating B then calls check_ready on the A-task.

Hence, we must either have the thread that updates nbor relations take care of updating both nbor lists "at the same time" (under lock), or we must make sure that an inconsistency does not have negative consequences.

If an nbor task (B) is present in the nbor list of a task (A), but that task (A) is not present in the nbor list of the nbor (B), then task (A) might find that it cannot update because B is not up-to-date, but when B updates it does not check if A becomes ready, because it is not on the nbor list. That problem could be handled by letting the addition of a new task to an nbor list always have the consequence to run check_nbors() on the nbor list (or at least to run check_ready() on the new nbor).

That simple fix might make it OK to delay the update of nbor lists until the thread that takes on an update can do it.

The implementation would then rely on these two mechanisms:

1) When adding or removing a task from the nbor list of the task a thread is

   working on, it must set a bit in the other task, triggering the next thread that starts to work on that task to update it nbor list accordingly; i.e., either remove the task has set the status bit, or add it. This means the actual nbor relation must match the action; if the action was "add", then the task must actually be a new, overlapping task, and if the action was "remove", then the task must actually either a) not exist any more, or b) have a flag set that indicates it is about to be removed.

2) After taking such an action, the thread must call check_ready() on a task that was added to an nbor list, and must run check_nbors() on the nbors of a task that is being removed – in case they will become updateable *because* the task is being removed. Again, the check_ready() must then recognize the bit that indicates a task is about to be removed

3.1.23 Nbor list renewal

Nbor lists are used for these purposes, in logical order

1) `list_t%check_nbors()` uses the nbor list to determine which nbor tasks to run `list_t%check_ready()` on. In `list_t%check_ready()` the nbor list is used to figure out if a task is ready to be updated, by comparing the states of tasks with the function `task_t%is_ahead_of()`.
2) After the task gets picked from the ready queue, e.g. by `dispatcher0_t%update`, the nbor list is used by `task_t%dnload` to pick up the information from its nbors that it needs, in order to update.
3) After the task has updated, and if it is a boundary task, the nbor list is used in `list_t%send_to_vnbors()`, which has as the main purpose to use MPI transparent / "invisble". To accomplish this, first of all the virtual copy of the active task must be updated, and then the `check_ready()` must be used on the nbor task, by a `check_nbors()` being executed on the virtual copy of the active task.

At which point can one abandone an existing nbor list, and adopt a new one, without creating a change of deadlock? Clearly, not after step 1), since the nbor lists used in `%dnload()` must be the same as the one used to check if the task was ready to update. Also, clearly not after step 2, since the nbor list at that point still carries information about which tasks were blocked by the active task not yet having been updated.

If a new task has been added by AMR, which happens between step 2 and 3, the requirements for being "ready" will change, for all nbors of the active task that will have the new task as a new nbor. All the previous nbors of the active task, and the active task itself, need to have new nbor lists generated.

Likewise, if a task is removed by AMR, all tasks in the nbor list of the active task need to have new nbor lists generated. The best point to do this is just after the task has updated, since if it is done before, there is no longer consistency between the nbor list and the task update.

With task addition, there is no problem doing the next `check_nbors()` with the old nbor list, since the new task by definition carries the same information as the parent task, and hence does not add any new constraint on "readiness", nor would adding the new task in the nbor list before the `dnload()` add any new information.

Both with task addition and with task removal, we require that the nbor list is generated by the thread assigned to update the task. By the reasoning above, this must be done at the very end of the update.

In the case of task removal, which is decided before the task update, there is no reason to actually remove the task until after it has updated; the update can by definition take place with the existing nbor list, and some nbors of the removed task have at the time already been added to the ready queue, and need access to the task as it was at the time.

Whether one removes the task before it is updated or after it has updated, one needs to trigger new nbor lists to be generated for all of the nbor tasks of the removed task. Those new nbor lists will not be generated until after those tasks have updated

3.1.24 Nbor list use

Nbor lists are used for these purposes, in logical order

1) In `list_t%check_ready()` the nbor list is used to figure out if a task is ready to be updated, by comparing the states of tasks with the function `task_t%is_ahead_of()`.
2) After the task gets picked from the ready queue, e.g. by `dispatcher0_t%update`, the nbor list is used by `task_t%dnload` to pick up the information from its nbors that it needs, in order to update.
3) After the task has updated, `list_t%check_nbors()` uses the nbor list to determine which nbor tasks to run `list_t%check_ready()` on.

   the nbor list of some task that depends on the active task is used to check if that task became ready to update after the current task updates. Such tasks are on the nbor list of the active task, and has a flag `needs_me` set.

3.1.25 Task handling

A new AMR task:

- isn't engaged in `check_ready()` of nearby tasks until it has been added to their nbor lists.
- also isn't involved in `check_ready()` performed by other threads until it has been added to the nbor lists of those nearby threads.
- is generally born ready to update, since it is formed as part of a parent patch that was about to be updated, and is given the same task time.
- may be added to the ready queue on an ad hoc basis, w/o necessarily having been given an nbor list at all
- does not need to be sent to the nbor ranks until after it has updated, at which time it will have a valid nbor list on the owner rank, and will be given one automatically – as for all new tasks on a node – when arriving on another rank

This argues in favor of delaying the call to `init_nbors (task%link)` until the next time it has been picked by a thread for updating. At that time, as part of the `refine_t%check_current()`, it needs to generate an nbor list, for use in `task%dnload`.

- At the time when it comes to calling `check_nbors`, it knows who the nbors are, but they may not yet know about the new task, and instead may rely on the still existing parent task.
- If the nbors don't have the new AMR task in their nbor lists yet, they cannot be denied updating based on the new AMR task, and thus at some point they will end up in the ready queue, and perhaps only then will they generate their own, new nbor lists
- but if a task generates a new nbor list after being selected, it may find that some of the new nbors may not be up-to-date, so this argues for generating new nbor lists at the end of a task update, rather than at the start
- except: a new AMR task needs to have a first nbor list, so it can get authoritative guard zone values. It is not necessary that the virtual copies get their nbor lists in any special way; they get theirs when created.

Using this strategy, we should be able to avoid calling `init_nbor_nbors()` after creating a new AMR task, and we should also be able to avoid calling `check_nbor_nbors()`, since the nbors will take care of their own.

3.2 Coding standard

The common coding standard has two separate purposes:

1. To ensure code consistency, which makes it easier to read and understand
2. Specific advantages when editing, such as easy of search, etc.

3.2.1 Modules

Modules should start with a description, with comment lines `!>` being automatically interpreted by `doxygen` at `readthedocs.org`:

```
!===============================================================================
!> Module description ...
!===============================================================================
MODULE some_mod
```

Then follows `USE` of the modules needed, and an `implicit none` (that applies also to all proceduress in the module):

```
USE io_mod
USE kinds_mod
USE ...
implicit none
```

All variables and procedures in the module should by default be private, with the exception of the data type definition, and a static instance of the data type, the purpose of which is to provide access to static parameters, set for example from a namelist, and given as default values to instances of the data type::

```
  private
  type, public:: some_t
    integer:: verbose=0
  contains
    procedure:: init
    procedure:: update
  end type
  type(some_t), public:: some
CONTAINS

!===============================================================================
!> Initialization
!===============================================================================
SUBROUTINE init (self)
  class(some_t):: self
  ...
END SUBROUTINE init

!===============================================================================
!> Update
!===============================================================================
SUBROUTINE update (self)
  class(some_t):: self
  ...
END SUBROUTINE update
...
END MODULE some_mod
```

3.2.2 Procedures

Procedures should follow this template, with a dotted line separating argument declarations from local variable declarations, and with dashed lines separating code and comment blocks::

```
!===============================================================================
!> Procedure description (picked up by doxygen)
!===============================================================================
SUBROUTINE updatet (self, aux)
  class(some_t):: self
  integer:: aux
  !.............................................................................
  integer:: local_variables, ...
  real(KindScalarVar), dimension(:,:,:), pointer:: d, ...
  real(KindScalarVar), dimension(:,:,:), allocatable:: w, ...
  !-----------------------------------------------------------------------------
  call trace%begin('some_t%update)
  ...
  !-----------------------------------------------------------------------------
  ! Comment block
  !-----------------------------------------------------------------------------
  ...
  var = expression                                   ! in-line comment
  ...
  call trace%end ()
END SUBROUTINE update
```

or, if the procedures should be timed::

```
!===============================================================================
!> Procedure description ...
!===============================================================================
SUBROUTINE updatet (self, aux)
  class(some_t):: self
  integer:: aux
  !.............................................................................
  integer:: local_variables, ...
  real(KindScalarVar), dimension(:,:,:):: d, ...
  integer, save:: itimer
  !-----------------------------------------------------------------------------
  call trace%begin('some_t%update, itimer=itimer)
  ...
  call trace%end (itimer)
END SUBROUTINE update
```

3.2.3 Capitalization

Capitalization of some keywords, such as:

```
MODULE ...
  USE xxx
  ...
CONTAINS

SUBROUTINE sub
END SUBROUTINE sub
```

(continues on next page)

(continued from previous page)

```
FUNCTION fun
END FUNCTION fun
```

serves the purpose of allowing editor search patterns such as `'USE xxx'`, `'E sub'`, or `'N fun'`.

3.2.4 Code

For consistency, and to maintain good readability even with many levels of indentation, code should be indented with (only) two characters per level, as in this template::

```
SUBROUTINE proc (self, arg1, arg2, ...)
  class(type_t):: self
  ...
  select type (arg1)
  class is (solver_t)
    n = ...
  class is (extras_t)
    n = ...
  class default
    n = ...
  end select
  do i=1,n
    a(i) = ...
    if (a(i) > 0.) then
      b(i) = ...
    else
      b(i) = ...
    end if
  end do
```

3.2.5 Comments

Comment in procedures should be either multi-line::

```
!-------------------------------------------------------------------------------
! The dashed lines should end in column 80
!-------------------------------------------------------------------------------
```

or single-line::

```
!--- This is a single line comment ---
```

Blank lines should be avoided (to maximize the visible code in editor windows), but if deemed absolutely essential to increase readability, they should have a leading comment sign, flush with surrounding lines (to allow fast skipping to the next procedure in editors)::

```
code ...
!
code ..
```

3.2.6 Debug printout

For largely historic reasons the code still contains a lot of debugging and diagnostic print messages, typically controlled by a local `verbose` parameter, similar to:

```
if (verbose > 0) then
  write (io_unit%log,'(....)') &
    wallclock(), ...
  if (verbose > 1) then
    ...
  end if
end if
```

Because many such code blocks can be very detrimental to code readability this is discouraged (meaning most of those constructs will be gradually removed), in favor of adding ad hoc statements of the type (fully left-adjusted, to stand out)::

```
write (stderr,*) wallclock(), mpi%rank, omp%thread, 'message', task%id
```

specifically tracing some action through various pieces of the code, to be removed after filling their function. The lines should be removed in a dedicated commit, so they are recoverable via a simple `git revert`, w/o introducing other changes.

3.3 Compilation

Compilation is controlled by a hierarchy of Makefiles. For each `experiments/whatever/` directory there should be a `Makefile` similar to those in parallel directories, so itechnical/f / when making a new experiment, use a `Makefile` from another experiment as a template.

Explanation:

The experiment `Makefile` does `include $(TOP)/config/Makefile`, which in turn includes the Makefiles in the various subdirectories, including the ones in the `config/compiler/` hierarchy, which determine compiler option settings.

Normally, it is not necessary to change the the make configuration, except possibly to overrule the choice of compiler, with for exampe `make COMPILER=ifort`.

See below for information on compiler option bundles, special make targets, and tailoring the make system:

3.3.1 make options

The default compiler options for gfortran and ifort are chosen as a compromise between compile time and performance; they generally give close to optimal performance, so are suitable for shorter runs and for development.

To get the best performance, compile with `make clean; make OPTS=optimized`, which typically gives a gain of a few percent in performance, at the cost of increased compilation time.

If the code crashes, try recompiling with `make clean; make OPTS=full_debug` or `make clean; make OPTS=debug`. The first provides comprehensive debug settings, while impacting performance significantly, while the 2nd avoids the particular options that impact performance the most.

The `make clean` part is only needed the 1st time after changing OPTS bundle

3.3.2 make targets

The `config/Makefile` specifies a few special target, which may be useful while developing code.

To see the values of various options (make macros= chosen, do `make OPTS=xxxx info`. Individual make macros listed may be overruled on the command line, using for example `make OPT=-O0` for a quick compilation; e.g. for checking syntax.

To get a list of source files compiled for a specific experiment, do `make source` (this requires that the code is compiled first). This may be particularly useful when looking for particular strings in the code; e.g. with:

```
make source | xargs egrep -n '(pattern1|pattern2)'
```

Some make macros are chosen based on for example the host name, or based on other macro values. To reveal where a particular option gets its values, do for example:

```
make showinclude PATTERN=COMPILER
make showinclude PATTERN=OPTS
make showinclude PATTERN=OPT
make showinclude PATTERN=FC
```

3.3.3 Makefile configuration

Several of the make options are chosen based on the value of the environment parameter `$HOST`. To influence those settings, create a file `config/hosts/$HOST`, and put the options you prefer there. A typical content on a cluster host would be:

```
COMPILER=ifort
```

You may choose to commit the file to the repository, but make sure to avoid collisions with existing files. Having the `$HOST` file in the repository makes sense on a cluster, to configure common settings used by several people, while committing a laptop `config/hosts/$HOST` to the repository is usually pointless.

To implement particular host-dependent options or option bundles for a compiler, create for example these files:

```
config/compiler/gfortran/$HOST/Makefile
config/compiler/gfortran/$HOST/optimized.mkf
config/compiler/gfortran/$HOST/debug.mkf
config/compiler/gfortran/$HOST/full_debug.mkf
config/compiler/gfortran/$HOST/personal.mkf
```

Note that these files only need to contain the lines that differ from the corresponding lines in the `config/compiler/gfortran/` files.

See also the note about `makes showinclude ...` on the page about **'make targets'_**

3.3.4 Separate builds

Especially when developing, it saves time to have separate `build_*/` directories; one for each combination of `SOLVER` and `OPTS`.

This may be enabled by setting the make macro `SEPARATE_BUILDS`, e.g. in a `$(TOP)/options.mkf` file (which might also be used to set the `COMPILER` and other macros). The file could thus contain, for example:

```
COMPILER=ifort
SEPARATE_BUILDS=on
```

3.4 dispatch.readthedocs.io

The dispatch.readthedocs.io documentation is generated automatically when pushing new commits to bitbucket.org/aanordlund/dispatch.git, but in order to proofread before pushing major updates it is very useful to generate a local cache. This is currently acthieved by doing, on a laptop:

```
cd docs
csh laptop.csh
```

This currently rsyncs the docs directory to astro08, and runs `make.csh` there, which uses the local Python installation to generate the HTML, and then rsyncs it to `ursa.astro.ku.dk/~aake/dispatch/`, so it may be accessed at `www.astro.ku.dk/~aake/dispatch/`.

NOTE: With a sufficiently complete local Python installation, the generation can be done locally, on any laptop, so the resulting HTML may be proofread there, by just doing (in the `docs` directory):

```
make html
```

3.5 Dust

Modeling the dynamics and growth of dust requires a convenient representation of the dust, and a convenient arrangement for computing the influence of the gas on the dust, and of the dust on the gas. These issues are addressed below.

3.5.1 Representation

To update the dust dynamics particle-by-particle is most easily done by keeping the particle information in a linked list (similar to the representation in the particle-in-cell plasma simulation setup.

We can use the same basic data type, extending it with the attributes that are particular to the dust-gas interaction.

Part of the representation is that the integer cell address is always known. The cost of computing the number density of each species is thus linear in the number of particles; as one moves the particles, their weights are summed up in each cell, and one thus arrives at their respective number density in each cell.

Given the number density of each species, one computes the rates of transformation from one species to the other using a kernel, and the resulting rates are summed up in each cell.

The next time the particles are moved over a certain time interval, their respective weights are changed according to the rates for that species.

3.6 Forces

Forces are now part of the extras_mod, so to add forcing take a copy of `extras/extras_mod.f90` into the `experiment/xxx/` dir, and uncomment the lines marked ... *! forces*

The new `forces_mod` does NOT collide with the old `force_mod` modules, so for compatibility one can construct new forces_mod modules by using `force%selected` to load values into `patch%force_per_unit_mass`.

However, in the future `forces_mod.f90` files should be written directly, since this alleviates the need to stick to a definitive (and long) argument list for the `force%selected` functions.

3.7 Google docs

Google docs allow documentation containing tables, drawing, graphics, and animations, and makes it easy to collect, edit and maintain such documents. This is much more efficient and simple than creating such documentation as part of the auto-generated HTML hierarchy, which is exclusively based on the GIT repository contents.

- Link to (for now private and partially out-dated) Google docs

3.8 GIT best practices

The Bitbucket web site has lots of advice on best practices for code maintenance, so here we just add advice specific to DISPATCH.

Note that many of the operations shown with command line syntax in the examples below may be easier carried out with GUI applications such as SourceTree.

3.8.1 Consolidating updates

When developing your project – either just adding input files and analysis scripts, or when doing actual development of new code – it is important to commit snapshots to your local working copy of DISPATCH often (essentially after each file edit), but also important to reorganize those commits before pushing to the on-line repository, which is typically a fork you may be sharing with collaborators.

The dilemma is that pushing all the small incremental commits that you make to the local working copy generates too much "noise" – both literally, in the form of emails, and in the sense that those commits are probably too fine-grained to be useful for others. On the other hand, refraining to make frequent commits obliviates one of the main advantages of code maintenance: the ability to "backtrack", to find out when and what went wrong, after the fact.

The way out of that dilemma is to reset to the start of the series of edits when done, after saving the detailed edits as a temporary branch (e.g. named yy-mm-dd for year-month-day), and then reorder and clean up, making a smaller number of well-commented commits before pushing those to Bitbucket repository.

One of the advantages with this is that debug statements that were first added and then removed disappear, reducing the "noise" in the commits that are pushed to the net.

Detailed examples:

Using rebase

```
# -- at start of the day --
git checkout master                          # get master branch
git pull                                     # pull in changes from elsewhere

# -- during the day --
... edit ...                                 # a small number of edits
git commit -a -m comment                     # GIT snapshot, with only terse comment
... edit ...                                 # a small number of edits
git commit -a -m comment                     # GIT snapshot, with only terse comment
...

# -- merge in edits from elsewhere --
git fetch                                    # get updates from elsewhere
git rebase origin/master                     # your edits become relative to that
```

(continues on next page)

(continued from previous page)

```
... may require some edits + commits ...     # in case of collisions

# -- consolidate --
git branch yy-mm-dd                          # create a backup, temporary branch
git reset origin/master                      # reset, to get all edits merged together
git add -p                                   # add _selected_, related edits
git commit -m "DBG comrehensive comment"     # possibly use 'commit -m' and an editor
git add -p                                   # add _selected_, related edits
git commit -m "ORG comrehensive comment"     # possibly use 'commit -m' and an editor
git add -p                                   # add _selected_, related edits
git commit -m "DIA comrehensive comment"     # possibly use 'commit -m' and an editor
git push                                     # push your consolidated commits
git branch -D yy-mm-dd                       # optionally, delete the temporary branch
```

3.8.2 GIT forks

A GIT fork is a GIT repository (e.g. at bitbucket.org), which is created as a copy of another repository, and which maintains a two-way connection, via which one can propagats updates both from and to the original.

Read about forks for example at the Bitbucket web site.

We suggest that you create a fork of the public repository on bitbucket.org, and work with private working copies of your fork, on laptops, as well as on clusters.

You may want (other) studentents to do the same or, alternatively, make a fork of your fork, to get an arrangement such as this one::

```
     private_version
          |         \
   public_version    private_forks
    /     |    |  \
 Ralph   Rolf |  Others...
   |          |
student   student
```

3.8.3 Logging

Log changes, showing statistics for each file::

```
git log --stat
```

Log changes, showing differences for each file::

```
git log --patch
```

Log the last 10 changes to a file, on all branches::

```
git log -n 10 --all --follow --stat -- dir/file
```

Show the last 10 changes to a file, on all branches::

```
git log -n 10 --all --follow --patch -- dir/file
```

3.8.4 Searching

Show the commit messages of files (in any branch) that contain "string"::

```
git log -G'\<string\>' --all
```

Show the commit messages of files (in any branch) that added or removed "string"::

```
git log -S'\<string\>' --pickaxe-regex --all
```

3.8.5 Rebase

To avoid a possibly confusuing git log that contains two parallel lines of developing, followed by a merge, it may be better to use `git rebase`.

To move your committed changes past new commits from a pull::

```
git rebase origin/master
```

Some merging of changes may be required after that

3.9 Object hierarchy

Main object hierarchy::

```
task_t                  ! Basic task attributes; time, position, ...
patch_t                 ! Mesh based task attributes; mesh, dimensions, ...
gpatch_t                ! A place to stick in IC value calls
extras_t                ! Layer with selectable, extra features
mhd_t                   ! MHD layer
solver_t                ! Generic layer, to present to experiments
experiment_t            ! Experiment layer, BCs, ICs, forces, ...
```

3.9.1 Call hierarchy

The basic call hierarchy, for quick reference::

```
dispatch
  mpi_mod::init                                  ! initialize MPI
    omp_mod::init                                ! initialize OMP
  io_mod::init                                   ! initialize I/O
  cartesian_mod::init                            ! initialize patches
    mpi_cords_mod::init                          ! determine MPI placement
    task_list_mod::init                          ! initialize task list
      list_mod::init                             ! initialize a list
    do
      patch_mod::set                             ! set bits
      patch_mod::init                            ! initialize patch
      task_list_mod::append                      ! append to task list
    end do
  task_list_mod::execute                         ! prepare and run the job
    list_mod::init_all_nbors                     ! initialize neighbor lists
```

(continues on next page)

(continued from previous page)

```
    list_mod::reset_status                          ! set boundary and virtual bits
    do
      task_mod::clear                               ! clear ready bit
      task_list_mod::check_ready                    ! check if patch is ready
        list_mod::queue_by_time                     ! add to ready queue
    end do
    tic                                             ! initialize performance counters
    timer_mod::print                                ! initialization time statistics
    do
      task_list_mod::update
        task_list_mod::check_flags                  ! check for flag files
        task_list_mod::check_mpi                    ! check for MPI messages
          mpi_mesh_mod::check                       ! check mesh lists
            MPI_IMPROBE
            MPI_GET_COUNT
            MPI_IRECV
            recv_list::add                          ! add to recv_list
            MPI_TEST
            unpack_list::add                        ! move to unpack_list
          mpi_mesh_mod::get                         ! get an incoming mesg
          task_list_mod::unpack                     ! handle unpacking
             task_list_mod::find_task               ! find the link with the task
             patch_mod::unpack                      ! patch unpack procedure
               anonymous_copy                       ! copy header into place
               patch_mod::header_to_patch           ! unpack header
               anonymous_copy                       ! copy mem into place
             deallocate                             ! deallocate mesg%buffer and mesg
        download_mod::download_link                 ! fill guard zones
        experiment_mod::output                      ! snapshots
        experiment_mod::update                      ! task update
        patch_mod::rotate                           ! time slot rotate
        task_mod::pack                              ! pack boundary patches
        link_mod::send_to_vnbors                    ! send to virtual neighbors
        task_mod::clear                             ! remove check bit
        list_mod::check_nbors                       ! check nbors and self
    end do
    timer_mod::print                                ! execution time statistics
    toc                                             ! performance cost summary
    buffered_output_mod::close                      ! close output files
  mpi_mod::end
```

3.9.2 Initialization calls

```
experiment_t%init
  solver_t%init
    mhd_t%init
      self%idx%init         (obsolete!)
      self%initial%init     (obsolete!)
      timestep%init         [only stagger2]
      gpatch_t%init
        self%idx%init       (new)
        self%initial%init
        self%idx%init       (new)
        patch_t%init
          task_t%init
```

(continues on next page)

(continued from previous page)

```
      force_t%init (obsolete!)
    extras_t%inig
      forces_t%init
        force_t%init
    validate%init
```

As much as possible should be inside framework files, avoiding requiring that all `$(SOLVER)%init` contain a chain of specific calls.

We should thus consider moving calls to `self%idx%init` to `gpatch_t`, and doing it both before and after the call to `self%initial%init`, as is done in stagger2 (to pick up changes of the `%mhd` switch).

Any calls to `self%initial%init` and `force%init` in `experiment_mod` files, or in `mhd_mod` files, should be considered obsolete.

3.10 jobcontrol.csh

The job script hierarchy consists of SUBMIT scripts, JOB scripts, and CANNED scripts.

The SUBMIT scipts contain scheduler control commands, but other otherwise static, since they are only read at submission.

The JOB scripts contain whatever the user wants, and may be changed during the runs – e.g. before restarting a process via jobcontrol.csh.

The CANNED scripts are helpful with for example archiving copies of executables, input files and log files.

3.10.1 Submit scripts:

```
long.sh
short.sh
...
```

These contains control syntax such as #PBS or #SBATCH. Since changes after submission have no effect, they call other scripts where changes do have effect. The call is via the jobcontrol.csh script, with submit script syntax such as

source $HOME/codes/dispatch/config/startup.sh jobcontrol.csh job.sh

The jobcontrol.csh script runs the script given as argument repeatedly, as long as there is a restart.flag file present in the run directory.

3.10.2 Job scripts:

```
long.csh
short.csh
```

These can be anything, including calls to standard scripts for local systems, with syntax such as

ivy.sh input.nml

3.10.3 Canned scripts:

```
config/host/steno/ivy.sh
```

Since we often want to use standard ways of handling executables, input files, and log file, we keep a number of canned scripts, such as ivy.sh, which places a copy of the excutable, the input file, and the log file, in the data/run/ directory.

The script seach path is setup by the config/startup.sh script, with directories seached in this order:

```
./
config/host/$HOST/
utilities/scripts/$SCHEDULER/
utilities/scripts/
$PATH
```

This meane one can override the default scripts by placing scripts with the same name (e.g. ivy.sh) in the run directory.

SCHEDULER is set by a file config/host/$HOST/env.{csh,sh}, if it exists, or may be set in the environment. Possible values are “pbs”, “slurm”, etc.

The utilities/scripts/$SCHEDULER/ directories contain scheduler-i specific scripts, used e.g. by jobcontrol.csh. Some examples are:

```
jobid             # return the id of the current job
jobkill id        # cancel a job
jobmail           # send a mail to $USER
```

3.11 Locking issues

3.11.1 Memory locking for guard zones

When `omp_lock%tasks` is set, external read access to task memory is locked while the task%update is actively modifying the task memory (this occupies only a small fraction of the time to do `task%update`), so the procedure is fully thread-safe. One can optimize this, at the cost of an extremely small chance to get relatively bad guard cell values on occasion:

When download_link wants to access a source task, it would do::

```
!------------------------------------------------------------------------------
! Acquire the task in locked mode, so nothing can change while we decide
! whether to keep it.  While we have the lock, we get the slot indices ``jt``
! and weights ``pt`` -- these are not going to change if we release the lock,
! and the source task updates -- unless it updates several times and those
! slots are overwritten with newer time values.
!------------------------------------------------------------------------------
call source%lock%set ('guard cells')
call source%time_slots (target%time, jt, pt)
!------------------------------------------------------------------------------
! If the target%time is in the first available time slot,
! which is source%t(source%iit(1:2)), that first slot could possibly be
! overwritten, in some very unusual cases, when the source task is just about
! to switch in that slot as "new", and if the current task by chance is delayed
! enough.  In that case, keep the lock until after guard zones are loaded.
!------------------------------------------------------------------------------
if (target%time < source%t(source%iit(2))) then
```

(continues on next page)

(continued from previous page)

```
  call target%get_guard_zones_from (source, jt, pt)
  call source%lock%unset ('guard cells')
else
  call source%lock%unset ('guard cells')
  call target%get_guard_zones_from (source, jt, pt)
end if
```

Note that the `get_guard_cells from (source)` procedure must rely on only the source%it information, and

3.11.2 Neighbor list synchronization

The link_t%nbor list is now changed nearly atomically, in that the new list is first constructed separately, and is then connected with a single instruction, pointing `link%nbor` to it. The procedure `list_t%init_nbors()` creates

1. a cached copy of the nbor list, which is used in `list_t%send_to_vnbors`
2. a cached copy sorted by level, which is used in `download_t%download_link()`

The nbor lists are also used in

3. `list_t%check_ready()`, which checks those nbors that have `nbor%needed` set, to see of their tasks are sufficiently advanced in time allow the linked task to be added to the ready queue
4. `list_t%check_nbors()`, which checks those nbors that have `nbor%is_needed` set, using `check_ready()` on each of them. This is done after every active and virtual task update, since those are the only points in time when new tasks could become ready to update

The task link, which contains `link%nbor`, must be protected by its OMP lock, `link%lock`, whenever the nbor list is updated or referenced. The cached nbor lists are used to keep these lock periods as short as possible

3.11.3 Lock timing

TODO: One could generalize the current pair of calls::

```
call object%lock%set ('label')
...
call object%lock%unset ('label')
```

to (with a meachnism very similar to `trace%begin / trace%end`):

```
integer, save:: ilock=0
...
call object%lock%set ('label', ilock=ilock)
...
call object%lock%unset (ilock)
```

and then inside `lock%set` measure the time it took to get the lock, and inside `lock%unset` measure the time the lock was held:

SUBROUTINE set (self, label, ilock) … lock%start(thread) = wallclock() … set lock .. lock%acquire(thread,ilock) = & lock%acquire(thread,ilock) + (wallclock()-lock%start(thread)) lock%start(thread) = wallclock()

SUBROUTINE unset (self, label, ilock) … … unset lock .. lock%held(thread,ilock) = & lock%held(thread,ilock) + (wallclock()-lock%start(thread))

3.11.4 OpenMP locking time and contention

A task typically has 26 nbors – possibly more in AMR situations. To get the guard cell values takes of order 10-30% percent to the update time, so at most of order 1% per source task.

To update the array memory takes a small fraction of the update time, so perhaps again of order 1% of the update time.

The life cycle of task has an period when the task is not busy, which typically is 50 times longer than the the task updat time. Hence the locking periods of time are extremely short in comparison to the task update cadence, and the chance the several target tasks are asking for the lock on a source while it is holding the lock is extremely small.

Hence, to conservatively lock the tasks while changing the task memory should give hardly any measurable impact on speed, while not doing it opens a small but non-zero change of memory conflict.

NOTE 1: The task does not need to prevent external memory access while doing things such as `task%pack` and `task%courant_condition`, since these procedures are performed by the thread that owns the task.

NOTE 2: On the other hand, while a virtual task is being unpacked, it should lock the memory, to prevent collisions with target tasks using it as source in guard cell loading.

3.11.5 Task mem array synchronization

During a task update, the only memory slot that is written to is `task_t%new`, while potentially all memory slots are needed to produce the new values.

Before a task update, all memory slots `iit(1:nt-1)` of nbor (source) tasks may in principle be needed, when computing guard zone values for the (target) task that the thread is going to update.

If the nbor task is dormant (with `bits%busy` not set), it could possibly start computing values after its time slot info has been acquired, but since the download for a single task takes less than 1% of the update, there is no chance whatsoever that the task has time to both finish one update, and start producing new mem array values for the next time slot (number 1 in terms of the iit array).

If the task is active (with `bits%busy` set), and is just about to finish its update, it might have time to do that, changing in the process its time slot indices and its task_t%time, but with no chance to finish one more update.

Hence, after retrieving (atomically) the time slot information from a source task, using the `task_t%timeslots()` procedure, the thread working on computing guard zone values for the target task can be sure that it can access slots (1:nt-1) without conflicting with updates to the source task.

The only possible exception would be if the task was suspended in the middle of the `download_t%different()` or `download_t%same()` procedure call, and did not wake up until the source task had updated several times. This might possibly (but still with very low probability) happen if one uses hyperthreading with a large factor (such as using 20 threads on 2 cores).

It is important, however, to make sure that the `source%t` and `souce%iit` values are retrieved only once, and that they are consistent, but if the local `iit()` values are based on a `source%it` that is updated as the last thing in `task_t%rotate()`, then even if the slot information is being changed by some thread executing `source%rotate()` during the target%timeslots() call, the target will still receive consistent `iit(1:nt-1)` memory slot values.

Consider the situation before the source update finishes::

```
iit(1) iit(2) iit(3) iit(4) iit(5)
  4      5      1      2      3
                      (it)   (new)
 t(1)   t(2)   t(3)   t(4)   t(5)
 0.3    0.4    0.0    0.1    0.2
 (1)    (it)  (new)   (4)    (5)
```

All that happens when the source update finishes is that `t(3)` is set to 0.5, the `new` mem slot is filled with updated variable values, and that the `iit(:)` array is shifted left, so it reads:

```
iit(1) iit(2) iit(3) iit(4) iit(5)
  5      1      2      3      4
                     (it)   (new)
```

The shift of iit(:) may be replicated by knowing only the value of `it`, which is set atomically. So, if `it` is incremented (cyclically) right after the time slot that it corresponds to has been updated, then a thread downloading to a target gets a set of times to interpolate between that is consistent with the content of the corresponding memory slots. Should the thread pick up `it` a microsecond too early it still gets useful data, since already the check_ready test that happened typically a long time ago made the judgement that the task was ready to be updated.

3.11.6 Locking timeline

Schematic timeline and call hierarchy showing lock set & unset:

```
task_t%update
  refine_t%check_current
    refine_needed
      need_to_support
        tlist%lock%set                                                          +|
        link%lock%set                                                  +|        |
        ..                                                              |        |
        link%lock%unset                                                -|        |
        tlist%lock%unset                                                        -|
    selective_refine
      make_child_patch
        parent%lock%set                                                +|
        parent%lock%unset                                              -|
        child%link%init
        child%init
          download_link
        tlist%lock%set                                                          +|
        tlist%append_link                                                        |
        tlist%init_nbors (parent%link)                                           |
          link%lock%set                                                          .
          do                                                                     .
            link%add_nbor_by_rank                                                .
            link%set_status_and_flags
          link%copy_nbor_list (... nbors)
          link%remove_nbor_list (... %nbors_by_level)
          link%sort_nbors_by_level (... %nbors_by_level)
          link%increment_needed (... %nbors_by_level)
          link%decrement_needed (nbors)
          link%remove_nbor_list (old_head)
        init_nbors (child_link)
           ditto
        init_nbor_nbors (child_link)
           link%lock%set
           self%init_nbors (link)                                      +|
           copy_nbor_list (link%nbor, nbors)                            |
           link%lock%unset                                             -|
           for nbor in link%nbor
             init_nbors (nbor%link)
           link%remove_nbor_list (nbors)
```

(continues on next page)

(continued from previous page)

```
        reset_status
        check_support()
          tlist%lock%set
          do link in tlist
            ...
          tlist%lock%unset
        send_to_vnbors
        check_nbor_nbors
          for nbor
            check_nbors (nbor%link)                                                    .
              ...                                                                      .
              check_ready                                                              .
          check_ready (link)                                                           |
        tlist%lock%unset                                                             -|
      remove_patch
    check_support(tlist,link)
  task%dnload
    download_link
      link%lock%set
      link%copy_nbor_list (nbors)
      link%increment_needed (nbors)
      link%lock%unset
      do
        source%lock%set
        same
        different
        source%lock%unset
      end do
      link%decrement_needed (nbors)
      link%remove_nbor_list (nbors)
  task%update
  task%rotate
  task%info
  send_to_vnbors
  check_nbors
    link%lock%set
    copy_nbor_list (... nbors)
    increment_needed (nbors)
    link%lock%unset
    for nbor ..
      check_ready (nbor%link)
        link%lock%set
        do .. nbor
          ..
        link%lock%unset
      queue_by_time (link)
        tlist%lock%set
  task%has_finished
```

3.11.7 Timings of relevance

In order to estimate the contention on nbor list locks it is useful to have estimates of the time spent in relevant routines.

Some of the central routines require approximately the times below, on a T430s laptop:

```
copy_nbor_list()   uses about 100 mus/call for 26 nbors, or about 4 mus/nbor
remove_nbor_list() uses about half the time, or about 2 mus/nbor

check_ready()      uses about 25 mus with 26 nbors, so about 1 mus/nbor
check_nbors(2)     uses about half that time, or about 0.5 mus/nbor
init_nbors()       uses about 1000 mus for 500 tasks, or about 2 mus/task
```

3.12 Maintenance

[This and related posts are mainly intended for the developers that maintain the development repository]

The maintenance instructions assume that the public and private repositories are checkout side-by-side, and are called *dispatch/{private,public}*. Other arrangements are handled correspondingly.

It is a good idea to add a local working copy of the private bitbucket repository as *upstream*, rather than the bitbucket repository itself – this reduces noise from the commits that would otherwise be needed.

3.12.1 4public branch

[This and related posts are mainly intended for the developers that maintain the development repository]

The *4public* branch in the development repository is used to communicate changes between the development repository and the public repository. To prepare for using it:

1. Start with a clean working copy of the public repository, connect the development repository as the *upstream* remote, and checkout the *4public* branch:

```
cd dispatch/public
git remote add upstream ../development
git fetch upstream
git checkout -b 4public upstream/4public
```

Make sure to never push the *4public* branch to the public repository, since that would create a very confusing situation. Always think of the *4public* branch as a branch in the development repository.

3.12.2 Common files

Occasionally, we may want to update all (or most) of the common files shared with the master branch in the development reposiory. This should not be done lightly, since it may brake existing codes, and (therefore) requires extensive testing.

To check differences and possibly pull in updates of common files, do::

```
# in the development working copy
cd dispatch/development
git checkout 4public
git pull
git diff --stat master -- `git ls-tree -r 4public --name-only`
... check carefully which files should really be included ...
git checkout master -- `git ls-tree -r 4public --name-only`
git checkout -- file1 file2      # cancel individual updates
... test carefully ...
git commit -m "... comrehensive comment ..."
```

(continues on next page)

(continued from previous page)

```
# in the public working copy
cd ../public                      # upstream = private
git checkout 4public              # branch connected to upstream/4public
git pull                          # pull in updates
git checkout beta                 # beta branch on public
git rebase master                 # sync with master on public
... test carefully ...            # test also that other experiments work
git commit --amend                # amend the commit message (cf. below!)
git push                          # push to private repository
```

In the amended commit message you should alert people about this new update of the `beta` branch, and invite them to try it out. Only after confirmation that there is no problem should the commit be carried over to the `master` branch of the public repository.

3.12.3 Pulling updates

To pull in updates from the development repository do::

```
cd dispatch/development           # working copy of private
git checkout 4public              # private branch
git pull                          # make sure it is up-to-date
git cherry-pick [ hash ]          # get the new feature
... test carefully ...            # test also that other experiments work

cd ../public                      # upstream = private
git checkout 4public              # branch connected to upstream/4public
git pull                          # pull in updates
git checkout beta                 # beta branch on public
git rebase master                 # sync with master on public
git cherry-pick 4public           # import feature
... test carefully ...            # test also that other experiments work
git commit --amend                # amend the commit message (cf. below!)
git push                          # push to private repository
```

In the amended commit message you should alert people about this new update on the `beta` branch, and invite them to try it out. Only after confirmation that there is no problem should the commit be cherry-picked over to the `master` branch on the public repository.

3.12.4 Pushing updates

To push updates from the private to the development repository do::

```
cd dispatch/development           # working copy of private
git checkout master               # must NOT be in 4public

cd ../public                      # upstream = private
git checkout 4public              # private branch
git pull                          # make sure it is up-to-date
git cherry-pick [ hash ]          # get the new feature
... test carefully ...            # test also that other experiments work
git push                          # push to private repository

cd dispatch/development           # working copy of private
```

(continues on next page)

(continued from previous page)

```
git checkout 4public                # branch connected to upstream/4public
git rebase master                   # sync with master on public
... test carefully ...              # test also that other experiments work
git checkout master                 # private master branch
git cherry-pick 4public             # import feature
git commit --amend                  # amend the commit message (cf. below!)
git push                            # push to private repos
```

Make clear in the amended commit message that this is imported from the public repos.

3.13 MPI

3.13.1 Overview

In order not ot be limited to use the initially existing task numbers when sending MPI messages, the most general recieving method uses `MPI_IMPROBE` to learn which task a message is aimed for, the size of the message, and its sequence number. After receiving a reply from `MPI_IMPROBE`, a buffer is allocated, a non-blocking `` `MPI_IMRECV `` is issued, and the message is added to a `recv_list` of outstanding receive messages. The messages in the list are checked regularly (once per task update), and when the message is complete the message object is moved to an `unpk_list` for unpacking.

A more efficient method, which should scale to any number of OpenMP threads, is to split communication between the threads, so each thread handles a limited set of virtual tasks, issuing an `MPI_IRECV` to each initially, and re-issuing an `MPI_IRECV` each time a package has been received. In most or all cases the packages then arrive in the order being sent, but if they do not, it is simple to add an out-of-order package to an unpack list, just as in the case above.

Similarily, send requests are started with `MPI_ISEND()` calls, and the requests are held on lists until each set of sends (each task is in general sent to multiple ranks) is completed (as tested via `MPI_TEST_ALL()`), at which time the send buffer is deallocated and the send request is removed from the `sent_list` and deleted.

It is an advantage to arrange this so that each thread maintain a thread- private `sent_list` – this avoids the need for OMP critical regions or locks.

3.13.2 Send procedures

Send requests are started with `MPI_ISEND()` calls, and the requests are held on lists until each set of sends (each task is in general sent to multiple ranks) is completed (as tested via `MPI_TEST_ALL()`), at which time the send buffer is deallocated and the send request is removed from the `sent_list` and deleted.

It is an advantage to arrange this so that each thread maintain a thread-private `sent_list` – this avoids the need for OMP critical regions or locks.

Data type and procedures::

```
task_mesg_t%check_mpi
  mpi_mesg_t%check_sent
    mesg_t%send
    mesg_t%test_all
    mesg_t%wait_all
```

3.13.3 Receive procedures

In order not to be limited to use the initially existing task numbers when sending MPI messages, the most general recieving method uses `MPI_IMPROBE` to learn which task a message is aimed for, the size of the message, and its sequence number. After receiving a reply from `MPI_IMPROBE`, a buffer is allocated, a non-blocking `` `MPI_IMRECV `` is issued, and the message is added to a `recv_list` of outstanding receive messages. The messages in the list are checked regularly (once per task update), and when the message is complete the message object is moved to an `unpk_list` for unpacking.

A more efficient method, implemented in `check_active()` and `check_virtual()`, which should scale to any number of OpenMP threads, is to split communication between the threads, so each thread handles a limited set of virtual tasks, issuing an `MPI_IRECV` to each initially, and re-issuing an `MPI_IRECV` each time a package has been received. In most or all cases the packages then arrive in the order being sent, but if they do not, it is simple to add an out-of-order package to an unpack list, just as in the case above.

Data type and procedures::

```
task_mesg_t%check_mpi
  task_mesg_t%check_priv
    task_mesg_t%unpack
    mpi_mesg_t%get
    mpi_mesg_t%add
    mpi_mesg_t%remove
    mpi_mesg_t%delete
    mesg_t%is_in_order
  task_mesg_t%check_virtual
    mesg_t%irecv
    mesg_t%is_complete
    mesg_t%is_in_order
```

3.13.4 Receiving order

A rank sends packages to several nbor ranks, so each rank also receives packages from several nbor ranks. It is important that these packages are received in the same order as they were sent from the sender rank, so the MPI protocol ensures that.

Each package sent from a rank to another rank contains a `tag`, which encodes the task ID and the sequence number. Each rank keeps track of the sequence number from other ranks in an atomic fashion. This is is handled by `task_mesg_t%check_priv()` and is detected when looping over the list of messages ready for unpacking. The procedure makes sure to call `task_mesg_t%unpack()` in sequential order. This is totally transparent to the underlying procedures (e.g. `patch_t%unpack()`, and detection of out-of-order messages is thus not needed there.

3.14 Python implementation

The technical details of the Python interfaced are described in the sub-pages below:

3.14.1 Python `dispatch` module

The `` `dispatch.snapshot()` `` procedure returns an object where the most important attribute is `` `snapshot.patches` ``, which is a list of patch objects (`` `p` ``), carrying attributes (e.g. `` `p.size` ``) that generally have the same name as the corresponding variables in the code.

Collecting patch metadata

The list of patches is collected from the `run/data/SSSSS/rank_RRRRR_patches.nml` files (here `SSSSS` is a 5-digit snapshot number and `RRRRR` is a 5-digit MPI rank). These files contain the metadata for all patches.

The actual data resides either in `data/run/snapshots.dat` (one file for all snapshots), or in `data/run/SSSSS/snapshot.dat` (one file per snapshot), or in `data/run/SSSSS/RRRRR_PPPPP.dat` (one file per patch).

Auxiliary data

The auxiliary data resides either in **data/run/SSSSS/RRRRR_PPPPP.aux** files (one file per patch).

To add data fields to it, register a link to 1-D, 2-D, 3-D, or 4-D data arrays, with::

```
USE aux_mod
...
type (aux_t):: aux
...
call aux%register ('name', array)
```

3.14.2 Expression handling

Expression handling should take place on four levels, in the most general case:

1. An arbitrary Pyhton expression is broken down into words, which may consist of * The left hand side of other expressions * Variables names * Python words – built-in or module objects
2. If evaluation of such words leads to no result the word is given to the var() function
3. The var function checks of the word belongs to known key-words, which respresent compound variables, such as 'T' for temperature, where each solver may require a different numerical expression, needing as much as 8 primitive variables (in the case of computing temperature when total energy is stored). * Expressions in the var function uses the mem() function to get actual values (which in fact or memory maps into disk files)
4. Inside the mem() function, alphabetic keys (such as 'px') are translated to integer indices, which in turn determine the offsets of the memory maps into the disk files

The first two steps take place in the expression_parser(), while the third step takes place in the var() function (possibly calling itself recursively), and the 4th step takes place in the mem() function.

In either the var() or mem() function, two aspect where the actual disk data may differ should be compensated for:

1. The different solvers use different centering of some of the variables
2. The io%format parameter determines, for example, if density is stored as log density, or as linear density.

The source code may be found in `utilities/python/dispatch/`, and is also available via the *auto-generated documentation* (the "Files" menu contains source code).

3.15 Radiative Transfer

Several radiative transfer solvers are implemented in the development version of DISPATCH, and based on that experience we are recommending the *short characteristics* method for general use, and including it in the public version.

The `experiments/stellar_atmosphere/` directory demonstrates the use of this solver in the context of stellar atmospheres. The method is general, however, with the choice of angles, opacity data, and boundary conditions defining the specific case.

3.15.1 Neighbor relations

The `nbors_t%init` procedure in the `solver/rt/$(RT_SOLVER)/nbors_mod.f90` file establishes the necessary nbor (dependency) lists and flags. It is called in this context::

```
!dispatcher_t%execute
!  rt_t%init_task_list (task_list)
!    rt_t%nbors%init
!      init_nbor_pair (task1, needs, task2, needs_me, download)
!    rt_t%prepend_tasks_to (task_list)
!      rt_t%init
!      task_list%prepend_link (rt%link)
!        task_list%prepend_link (rt%omega%link)
!
```

3.15.2 Task list manipulation

The task list is first constructed by the `component/` procedure setting up the arrangements of MHD tasks. In the common case with a Cartesian arrangements of `patch_t` tasks, the relevant file is `components/cartesisan_mod.f90`. The sequence of calls that happen are::

```
!cartesian%init
!  task_list%init
!    experiment_t%init
!    task_list%append (experiment)
```

which results in a global `task_list` containing only the exeperiment tasks, which in this case are `solver_t` tasks, where the task structure has been extended in `extras_t%init` with a `solver_t%rt` RT task, which in turn has extended itself with a set of *´solver_t%rt%omega(:)‘* tasks, one for each ray direction. These extra tasks are referred to as "RT sub-tasks", and have not yet been added to the task list, since `cartesian_t%init()` only adds the `experiment_t` tasks (cf. above).

To give all tasks access to the task list, before calling the specific `dispatcher_t%method()` the `dispatcher_t%excute` procedures calls each task with:

```
!call task%init_task_list (task_list)
```

This provides the opportunity to add additional tasks to the tasks list (to avoid a potential recursive disaster the new tasks are prepended rather than appended). In the RT case, the `rt_t%init_task_list(self,task_list)` is first adds the RT sub-tasks to the task list and then sets up the proper nbor relations between the new tasks and the already existing MHD tasks.

3.15.3 Short characteristics

Short characteristics radiative transfer solvers typically solve the radiative transfer equations across only a single cell (or in some cases even a fraction of a cell) at a time. In three dimensions one can nevertheless achieve excellent performance, by parallelizing over perpendicular directions – effectively solving for the radiation over parallel planes, progressing from one plane to the next, starting from a boundary where values are known, either from physical boundary conditions, or from boundary values taken from an adjacent ("up-stream") domain.

Because one is looping over two redundant directions, it is possible to significantly reduce the cost, since it allows the compiler to use loop vectorization.

The `solvers/rt/short_characteristics/` directory contains the following modules, used to perform various parts of such solutions::

```
radau_mod.f90                 ! Radau integration -- a modified Gauss integration
rt_integral_mod.f90           ! integral method solver
rt_mod.f90                    ! RT data type definitions
rt_nbors.f90                  ! RT neighbor setup
rt_solver_mod.f90             ! RT solver data type
```

Radau integration

Radau integration differs from normal Gauss integration only in that one of the interval end points is always included. Here, the integration is performed over $\mu =$ the cosine of the inclination relative to the normal, and the point $\mu = 1$ is always included.

This has the advantage that no ϕ-integration (integration over azimuth) is needed for that inclination, which also generally carries the largest specific radiation intensity.

Scheduling

Radiative transfer tasks need not be updated with the same cadence as their MHD "host" tasks. The scheduling relations are illustrated below.

Phase 1: MHD updates are done, until `mhd%time >= mhd%rt_next` (=`omega%time` for all omega)::

```
------------- BC --------------------------------------
------------- MHD1 ------------+  |
--------------RT1  ---------------+
                                  |
------------- MHD2 -------------+ |
--------------RT2  ---------------+
```

Phase 2: EOS2 calculations, at `time = mhd%rt_next`, setting `eos_time` to this time. Here, EOS2 has advanced, and with MHD2 time ahead, only lack of upstream RT prevents RT2 update:

```
------------- BC --------------------------------------
------------- MHD1 -------------+ |
--------------RT1  ---------------+
                                  |
------------- MHD2 ---------------|-+
--------------EOS2 ---------------+
--------------RT2  ---------------+
```

Phase 3: EOS1 calculations, at `time = mhd%rt_next`, setting `eos_time` to this time. Here, EOS2 has advanced, and with MHD2 time ahead, only lack of upstream RT prevents RT2 update:

```
------------- BC --------------------------------------
------------- MHD1 ---------------|+
--------------RT1  ---------------+
                                  |
------------- MHD2 ---------------|-+
--------------EOS2 ---------------+
--------------RT2  ---------------+
```

Phase 4: RT1 is now detected as "ready", while RT2 is waiting:

```
------------- BC --------------------------------------
------------- MHD1 ---------------|+
--------------RT1  ---------------|--------+
                                  |
------------- MHD2 ---------------|-+
--------------EOS2 ---------------+
--------------RT2  ---------------|--------+
```

If the RT update time `rt_time` is *smaller* than `mhd%dtime`, it could happen that the next time the MHD task is updated, it finds itself still ahead of the RT time, and hence that MHD task (but not all of them, advances it's RT time again).

A simple solution would be to limit the MHD update time interval to `rt_time`, for all patches that do RT. This could impact the cost a bit, but only when the choice of RT time is made very conservative

Tasks

This is what the class hierarchy looks like when radiative transfer (RT) is included::

```
!experiment_t -> solver_t -> mhd_t -> extras_t  -> gpatch_t -> patch_t -> task_t
!                                     |---rt_t -> gpatch_t -> patch_t -> task_t
!                                     |-grav_t -> gpatch_t -> patch_t -> task_t
```

The task list consist of 1) a number of "MHD tasks" (`experiment_t` tasks), and 2) A number of `rt_t` tasks, which subdivide into 2a) one "main RT task" (an `rt_t` task) for each MHD task, and 2b) `n_omega` "RT sub-tasks" (also `rt_t` tasks)

Each MHD task has connections to the `rt_t` tasks via the `extras_t` class, and each main RT task has connections to the RT sub-task, which are allocated as an array of `rt_t` tasks inside the main `rt_t` task. Each `rt_t` task also has a connection to the patch_t part of the MHD task, and can thus access the MHD time and other attributes

The MHD task update procedure is called directly from task_list_t%update(), and calls extras_t%pre_update before the MHD update, and extras_t%post_update after it. Through that call path, `rt_t%post_update` is (only) called when the MHD task has updated. It is not called as part of the normal `rt_t%update`. This is the correct point to detect that a new EOS calculation is needed.

The RT tasks are also called directly from the `task_list_t%update()`, and as illustrated above, they do not go through the `extras_t` layer. Those calls go only to `rt_t%update()`.

3.15.4 Point source radiation

Point source radiative transfer is available in the development repository, on a collaborative basis. Contact aake@nbi.ku.dk if you are interested in this.

3.16 Sink-particles

A sinkparticle is represented by a task data type (`sink_patch_t`) that is primarily an extension of a `gpatch_t` data type, with a `patch%mem` that has a limited extent (~8 cell radius). Sink particles are formed at the same resolution level as the patch they are formed in.

From the point of view of the dispatcher task_list, the task appears as a normal patch, but with special case handling that causes all interior values to be downloaded from and to its normal MHD patch nbors. The particle aspect of a sinkparticle is kept in a `partcle_list_t` data type, which stores several positions in time for the particle, making it possible to interpolate its position to any given time, when computing interactions (forces).

The sink particle position is updated by an N-body solver, which gets access to the other sink particle histories by being an extension on top of the `task_list_t` data type.

The particle position update method is always one particle at a time, since particle updates use variable time steps, which differ from particle to particle. A particle "update" in the DISPATCH context thus consists of these steps

1) pre_update: estimate the mid-point position to use, in order to make the update time-reflexive
2) compute the forces at some point in time (e.g. the initial time for the 1st K step in KDK) and update the velocity
3) update the position (e.g. with D step in KDK)
4) compute the forces at some other point in time (e.g. at final time for 2nd K step in KDK) and update the velocity

3.16.1 Ad hoc sink particles

To allow testing, one can create *ad hoc* sink particles, using namelist entries similar to:

```
&sink_patch_params       verbose=2 on=t n_ad_hoc=2 /
&sink_params             x=0.52 y=0.52 z=0.52 vx=0.1 mass=1 /
&sink_params             x=0.48 y=0.48 z=0.48 vx=0.2 mass=1 /
```

Whereas sink particles normally are created and added to the task list by one of the refinement criteria, the *ad hoc* particles are created after the task list with normal MHD patches has been created, via a call from `extras_t%init_task_list`, which gets called once for every patch, just before updates of the task list start.

The call has the current task list as an argument, which makes it possible to append the new tasks, corresponding to the *ad hoc* sink particles.

3.16.2 Forces

The forces that involve sink particles are of two types:

1. Gravitational forces between particles. These can be computed very efficiently, by first caching the particles from list form to array form, and then using vectorization over the particles. The cost per particle is low enough to accept, for each particle, direct computation for hundreds of the most significant sink particles. Each force computation is used to accumulate both the force of A on B, and the force of B on A.
2. Gravitational forces between gas and particles may, likewise, be computed by direct summation over particles, costing of order a few nanoseconds per sink particle, and thus allowing hundreds of sink particles w/o more than doubling the cost per cell.

   As with the particle-particle force, one can accumulate, without additional cost, the force of the gas on the particles, from the action = reaction principle. This yields for each patch and time, a force field from all the direct summation sink particles.

   To use this information in the calculation of the forces acting on each particle requires (only) that one interpolates in time, for each patch. For each patch we know the force on each particle, at a discrete set of times. From that information one can interpolate to the exact particle time, using high order time interpolation (`lagrange_mod.f90`).

   To quickly find the right `patch_force_t` instance, given the particle and MHD patch IDs, we use a hash table with `(particle%id,patch%id)` as a 2-dimensional key, and an anonymous pointer with the address to the `patch_force_t` data type as value.

3.16.3 Data type hierarchy

The `sink_patch_t` data type is an extenstion of the standard `gpatch_t` data type, with access to the task list.

The `particle_solver_t` is an extension of the `particle_list_t` data type, which is a ‘‘ task_t‘‘ extension that holds particle positions at several times. The `particle_solver_t` is kept as an attribute of ‘‘ sink_patch_t‘‘ data type, so the data type hierarchy looks like this:

```
                           task_list_t
                            / |  |
                experiment_t  |  |
                 solver_t     |  |
                   mhd_t      |  |
                     | refine_t  |
                     |  /        |
                  extras_t       |
                     |           |
               sink_patch_t      |
              /      |           |
    paricle_solver_t |           |
             |       |           |
             |     gpatch_t      |
             |    /  |     \     |
             |   /   |      list_t
     particle_list_t |     /  |
      /      |      patch_t   |
particle_t dll_t    /   \     |
                   /     link_t
                  /       |
            connect_t   task_t
```

3.16.4 Reflexive time steps

To make the particle evolution near-symplectic the most important aspect is that the time step determination should be *reflexive*, in the sense that when taking a timestep forward from `t_A` to `t_B`, the timestep should be evaluated in such a way that starting from `t_B` and moving backwards in time one should end up exactly at `t_A`. (Aiming for exact symplectic expressions would be overkill when there are weak and non-conservative perturbations from the moving gas).

A simple and efficient approximate method to achive this is to extrapolate the position forward one half (time index) step, using the previous positions, which are stored in the particle history. Details of the extrapolation may differ, e.g. in using or ignoring speed information – the main goal should be to obtain an estimate that is both cheap and accurate.

3.16.5 Task update sequence

When a sink particle task (data type `sink_patch_t`) reaches the head of the ready queue and is taken by one of the threads, its call to the normal `task_list_t%update()` procedure results in calls to these procedures (indentation indicates call level, and the name of the file containing the procedure is obtained with the substituttion `_t` -> `_mod.f90`):

```
experiment_t%dnload                                          ! generic download call
  sink_task_t%dnload                                         ! sink patch download
    download_t%download_link (..., all_cells=.true., ...)    ! values for accretion
task%update                                                  ! generic update call
```

(continues on next page)

(continued from previous page)

```
  sink_task%update                                         ! sink patch update
    particle_solver_t%force_field                          ! fall through
      particle_list_t%force_field                          ! compute forces
        hash_table%get                                     ! get patch_forces
    sink_task_t%accrete                                    ! accrete mass
      sink_task_t%courant_condition                        ! set timestep
    sink_task_t%move                                       ! organize particle move
      particle_solver_t%update                             ! particle solver update
        particle_solver_t%courant_time                     ! particle courant time
patch_t%rotate                                             ! patch periodicty etc
  task_t%rotate                                            ! rotate time slots
list_t%send_to_vnbors [ if the task is a boundary task ]   ! send boundary tasks
```

3.16.6 Stellar winds

The procedure to take the stellar wind from a sink particle into account should be an extras procedure, executed by the %update procedure of the patch that is receiving the input of mass and momentum from the sink particle.

3.17 Stationary Lagrangian

The *Stationary Lagrangian Method* combines Eulerian mesh placement and Lagrangian dynamics, by making a shift in velocity space that cancels most of the bulk motion.

3.17.1 Refinement considerations

One could choose to refine from either the %it state – aligned with the mesh, but with velocities shifted by %x_shift

3.18 Task lists

Tasks lists are fundamental to DISPATCH, and the list nodes (data type link_t) contain several types of pointers that define relations between tasks; e.g. a subset with time order, or a subset of active tasks.

The task list should not be required to know any specifics of tasks, which are referred to with pointers of type experiment_t; the anonymous top tier task, which may be anything between the top of a hierarchy with many layers (e.g. task_t -> patch_t -> gpatch_t -> mhd_t -> rmhd_t -> solver_t), or just two layers (task_t -> experiment_t).

The task_list_t data type extends the list_t data type with procedures that can handle tasks in the context of task lists; e.g. specifying what happens in the context of unpacking a task MPI package from another MPI process.

3.18.1 Procedures

As is, the task_list_t data type contains a number of procedures that have to do with message sending and receiving. These could possibly be split off into task_mesg_t data type. Alternatively, the procedures that in effect implement dispatcher method=0 could be split off into a dispatcher_method0_t data type, leaving task_list_t to effectively be the task_mesg_t.

The list_t data type contains (or should contain only) procedures that manipulate lists of link_t data types; inserting or deleting nodes in lists, etc.

As is, the list_t data type also contains procedures that rely on tasks having meshes, e.g. for constructing neighbor lists. These could perhaps with advantage be split off into a patch_list_t data type.

tasks, one should be able to define task type specific relations that define when a source task is ahead of a target task. This requires the functions that call is_ahead_of are at a level where they are aware of experiment_t and all sub-levels.:

```
dispatcher_t                                                dispatcherX_mod
  task_list_t check_ready                                   task_list_mod
    experiment_t is_ahead_of refine                         experiment_mod
      solver_t                                              solver_mod
        gpatch_t                                            gpatch_mod
          patch_t                                           patch_mod
            task_t                                          task_mod
```

For task refinement, a similar desire exists. The procedure that defines if a task should be "refined" (whatever that means) should be aware of all levels of the hierarchy. Or else, one should be able to overload the refine procedure itself, at any level.

3.18.2 Refinement

How should refinement procedures and the dispatchers really work together? Currently, the task list updater calls are refine procedure, as part of the task_list_t%update procedure, but this is awkward, since it means that one allows a procedure that really deals with a single link in a task list to affect the task list itself. It would be safer and more consistent if the level that loops over task list events (i.e., the dispatcher level) is the one that also considers refinement.

But if the refine_mo should be able to manipulate the task list with dispather0, it needs to know about task list, which creates a Catch 22, since it is also called from inside task_list_mod.:

```
dispatcher_t                                                dispatcherX_mod
task_list_t                                                 task_list_mod
list_t                            check_ready               list_mod
              experiment_t                                  experiment_mod
              rt_solver_t                                   rt_solver_mod
              rt_t               (is_ahead_of)              rt_mod
refine_t ---------------------------------------------------- refine_mod
              solver_t              |                       solver_mod
              mhd_t                 |                       mhd_mod
              gpatch_t              |                       gpatch_mod
              patch_t               |                       patch_mod
link_t                              |                       link_mod
              task_t              is_ahead_of               task_mod
```

3.19 Task locking

DISPATCH uses OpenMP nested locks, with the API defined in `omp/omp_locks_mod.f90`.

A basic rule that needs to be respected with locks is to avoid that two objects of the same class are locked at the same time by a task – the situation below can clearly lead to a deadlock

```
thread 1:
  lock A(1)
    lock & unlock A(2)
  unlock A(1)
thread 2:
```

(continues on next page)

(continued from previous page)

```
lock A(2)
  lock & unlock A(1)
unlock A(2)
```

On the other hand, if a thread locks one type of lock, and then a whole set of another type of locks inside the first one, this cannot lead to a deadlock, since either the outer locks are the same, and only one thread can do the set, or they are different, and they can negotiate. Hence locking the task list avoids any deadlock that could potentially be triggered if threads were not locked out.

3.20 Timeslots

Each task has associated with it two sets of information that are also accessed by other threads:

1) the `task%t(:)`, `task%dt(:)`, `task%time`, and `task%it` (equivalently `task%iit`) info about time slots
2) the `link%nbor` nbor lists, which lists the nbor tasks that the task depends on

The 1st set of values are accessed twice: First in `list_t%check_nbors()` and `check_ready()`, which uses task_t%time and task_t%dtime to determine if the nbors of a task are sufficiently advanced in time to consider the task ready to update.

Later, when the guard zone values are downloaded, all of the first `(nt-1)` values of `task_t%t(:)` and `task_t%dt(:)` are needed.

The 2nd set of values (or pointers) are used in both of these steps as well. However, since the nbor lists are not changed, or else are cached, there is no problem or need for locking in this context, except very briefly, while making the sorted cache copy and, correspondingly, when changing the nbor lists in connection with refine / derefine.

CHAPTER 4

Auto-generated documentation

The code is documented with internal comment blocks such as this one:

```
!=======================================================================
!> Comment test
!=======================================================================
```

at the top of modules and module procedures. Such comment blocks are processed by Doxygen at ReadTheDocs, and turned into a set of HTML pages, showing the relations between modules, data types, and module data and procedures:

- DISPATCH Doxygen output