LIBPFASST

Release 0.5.0

Feb 28, 2018

Contents

The **Parallel Full Approximation Scheme in Space and Time** (PFASST) algorithm is a method for parallelizing ODEs and PDEs in time. The maths behind the PFASST algorithm are breifly described on the *Maths* page.

This work was supported by the Director, DOE Office of Science, Office of Advanced Scientific Computing Research, Office of Mathematics, Information, and Computational Sciences, Applied Mathematical Sciences Program, under contract DE-SC0004011. This work is currently authored by Michael L. Minion and Matthew Emmett. Contributions are welcome – please contact Matthew Emmett.

CHAPTER 1

LIBPFASST

LIBPFASST is a Fortran implementation of the PFASST algorithm.

Main parts of the documentation

- *Download* - download and installation instructions.
- *Tutorial* - getting started and basic usage.
- *Overview* - design and interface overview.
- *Reference* - information about the internals of LIBPFASST.
- *Communicators* - information about the various communicators available.

1.1 Contributing

LIBPFASST is released under the GNU GPL, please see the LICENSE file in the source code repository for more information.

If you would like to contribute to the development of LIBPFASST, please, dive right in by visiting the LIBPFASST project page.

1.1.1 Maths

The Parallel Full Approximation Scheme in Spact and Time (PFASST) algorithm is time-parallel (across the domain) algorithm for marching time-dependent ODEs or PDEs through time.

The PFASST algorithm is perhaps most easily understood as a parallel version of the multi-level Spectral Deferred Correction (MLSDC) scheme (see below). Some aspects of PFASST are also reminiscent of the Parareal algorithm (see below).

MLSDC

XXX

Parareal

The Parareal method can be roughly described in terms of two numerical approximation methods, denoted here by $\mathcal{G}$ and $\mathcal{F}$. Both $\mathcal{G}$ and $\mathcal{F}$ propagate an initial value $U_n \approx u(t_n)$ by approximating the solution to

$$\dot{u} = f(u,t), \quad u(t_n) = u_n$$

from t_n to t_{n+1}. For the methods to be efficient, it must be the case that the $\mathcal{G}$ propagator is computationally less expensive than the $\mathcal{F}$ propagator, and hence $\mathcal{G}$ is typically a low-order method. Since the overall accuracy of the methods is limited by the accuracy of the $\mathcal{F}$ propagator, $\mathcal{F}$ is typically higher-order and in addition may use a smaller time step than $\mathcal{G}$. For these reasons, $\mathcal{G}$ is referred to as the coarse propagator and $\mathcal{F}$ the fine propagator.

The parareal method begins by computing a first approximation U^0_{n+1} for $n = 0 \ldots N-1$ where N is the number of time steps, often performed with the coarse propagator:

$$U^0_{n+1} = \mathcal{G}(t_{n+1}, t_n, U^0_n).$$

The Parareal method then proceeds iteratively, alternating between the parallel computation of $\mathcal{F}(t_{n+1}, t_n, U^k_n)$ and an update of the initial conditions at each processor of the form

$$U^{k+1}_{n+1} = \mathcal{G}(t_{n+1}, t_n, U^{k+1}_n) + \mathcal{F}(t_{n+1}, t_n, U^k_n) - \mathcal{G}(t_{n+1}, t_n, U^k_n)$$

for $n = 0 \ldots N-1$. That is, the fine propagator is used to refine the solution in each time slice in parallel, while the coarse propagator is used to propagate the refinements performed by the fine propagator through time to later processors.

The PFASST method operaters in a similar manner to the Parareal method, but it employs the MLSDC time-integration technique in a novel way to improve the parallel efficiency of the method.

References

1.1.2 Download

You can obtain a copy of LIBPFASST through the LIBPFASST project page on bitbucket.org. Complilation instructions can be found in the README file included with LIBPFASST.

1.1.3 Tutorial

Please see the **'mpi-advection'_** example included in LIBPFASST for a simple application of LIBPFASST.

This example solves a 1d linear advection diffusion equation

$$u_t + v u_x = \nu u_{xx}.$$

This equation will be split into stiff (νu_{xx}) and non-stiff ($-v u_x$) pieces, and hence an IMEX SDC substepper will be used to evolve the equation through time.

The main program is in `src/main.f90`. The IMEX sweeper needs three functions: one to evaluate the non-stiff piece, another to evaluate the stiff piece, and one more to solve a backward-Euler step. These routines are in `src/feval.f90`.

Three PFASST levels will be used. The coarsest level will have 2 SDC nodes and 32 spatial points; the intermediate level will have 3 SDC nodes and 64 spatial points; and the fine level will have 5 SDC nodes and 128 spatial points. The routines to spatially interpolate and restrict solutions are in `src/transfer.f90`. Two coarse sweeps will be performed per iteration on the coarsest levels. This helps reduce the total number of PFASST iterations required.

The solution u will be stored in a flat Fortran array, and hence this application will use LIBPFASSTs built in `ndarray` encapsulation. Note that LIBPFASST doesn't impose any particular storage format on your solver – instead, you must tell LIBPFASST how to perform a few basic operations on your solution (eg, how to create solutions, perform `y <- y + a x`, etc). Various hooks are added to echo the error (on the finest level) and residual (on all levels) throughout the algorithm. These hooks are in `src/hooks.f90`.

Note that the `feval_create_workspace` routine is specific to the problem being solved (ie, not part of LIBPFASST, but part of the user code). It creates FFTW plans, creates a complex workspace for use with FFTW (so that we can avoid allocating and deallocating these workspaces during each call to the function evaluation routines), and pre-computes various spectral operators.

LIBPFASST allows you, the user, to attach an arbitrary C pointer to each PFASST level. This is called a context (typically called `levelctx` in the source) pointer (as in, "for the problem I'm solving I have a specific context that I will be working in"). Your context pointer gets passed to your function evaluation routines and to your transfer routines. In most of the examples this context pointer is used to hold FFTW plans etc as described above. Note that each level gets its own context because each level has a different number of degrees-of-freedom (`nvars`).

C pointers are used because they provide a lot of flexibility. The drawback to this is that we loose the ability for the compiler to do type checking for us.

1.1.4 Overview

The LIBPFASST library evolves systems of ODEs in time:

$$u'(t) = f(u(t), t).$$

LIBPFASST comes with builtin sub-steppers for fully explicit, fully implicit, and implicit-explicit (IMEX) systems.

The user provides:

1. Function evaluation routines: $F(U, t)$. These functions may depend on the PFASST level, which allows for multi-order and/or multi-physics operators.
2. For implicit or IMEX schemes, a backward Euler solver: $U - dtF(U) = RHS$.
3. Spatial interpolation and restriction routines: $R(U)$ and $P(U)$.
4. Solution encapsulation routines to tell LIBPFASST how to manipulate solutions. This enables LIBPFASST to interoperate nicely with other libraries that have particular ways of storing solutions (eg, BoxLib, PEPC, or PETSc). In particular, this allows LIBPFASST to work with spatially distributed solutions.

User interactions with LIBPFASST are typically marshaled through the `type(pf_pfasst_t)` class. This class acts as the overall controller of the algorithm. Implementing a ODE/PDE solver in LIBPFASST generally consists of the following steps:

1. Create an `type(pf_pfasst_t)` controller.
2. Create an `type(pf_encap_t)` encapsulation object. This may be level dependent.
3. Create an `type(pf_comm_t)` communicator object. This allows LIBPFASST to use MPI or pthreads parallelization (independent of any spatial parallelization).
4. Create an `type(pf_sweeper_t)` SDC sweeper. Builtin sweepers include: explicit, implicit, or imex. Custom sweepers can also be built. This can also be level dependent.

5. Add levels to the `type(pf_pfasst_t)` controller. This includes telling LIBPFASST how many degrees-of-freedom are on each level, how many SDC nodes should be used on each level and their type, which interpolation and restriction routines to use, which encapsulation object to use etc.
6. Add any desired hooks to the controller and set any controller options.
7. Call the communicator's and controller's setup routines.
8. Run.
9. Tidy up.

1.1.5 Reference

LIBPFASST is, for the most part, written in Fortran 90. However, it also uses procedure pointers and the ISO C binding module from Fortran 2003.

Please see *Parameters* for a list of PFASST parameters.

1.1.6 LIBPFASST communicators

LIBPFASST includes several (time) communicators:

- MPI
- pthreads
- fake

The MPI communicator is the most actively developed and maintained. The pthreads communicator exists for performance testing purposes and may be deprecated in the future. The fake communicator is used for convergence testing (convergence tests across many 'fake' cores can be performed on one core).

The number time steps taken N should be an integer multiple of the number of PFASST processors P used. If more time steps are taken then PFASST processors used, the default behaviour of LIBPFASST is to operate in `PF_WINDOW_BLOCK` mode.

In `PF_WINDOW_BLOCK` mode, P time steps are taken at a time. Once these P steps are completed, the final solution on the last time processor is broadcast to all time processors (and is received as a new initial condition) and the PFASST algorithm is started over (including the predictor stage) again. This process is repeated N/P times until all N time steps are complete. This means that the first processor may be idle for a time while it waits for the last processor to finish (due to the burn in time associated with the predictor step) after each block of P time steps are computed.

Alternatively, in `PF_WINDOW_RING` mode, when a processor is finished iterating on the time step that it is operating on, it moves itself to one time step beyond the last processor.

At the beginning of each PFASST iteration (before receive requests are posted) each processor: (i) determines if it has converged by checking residuals, and (ii) receives a status message (this is blocking so that the status send/recv stage is done in serial). This status message contains two pieces of information: whether the previous processor is finished iterating (in which case the previous processor will move itself to the end of the line), and how many of of the previous processors are going to move to the end of the line.

If all residual conditions are met AND the previous processor is done iterating, the current processor will decide that it is done iterating too. It will then send this information to the next processor (and bump up the number of processors that are going to move by one).

If a processor is done iterating, then it will move to the end of the line. To do this, it does a blocking probe for a message with the special "nmoved" tag and with source `MPI_ANY_SOURCE`. This special message is sent by the first processor that is still iterating *and* whose previous processor moved, **or** by the last processor in the event that all

processors finished at the same iteration. In this case, even though the last processor is done iterating, it will do one final iteration before moving so that normal communication can proceed.

In any case, all processors rotate their first/last markers by the number of processors that moved.

Finally, if a processor steps beyond the number of requested time steps it stop iterating. In this case, other processors move their last marker accordingly so that they do not include stopped processors in their communications.

MPI

The PFASST MPI communicator can be used in conjunction with spatial MPI communications. The user is responsible for splitting `MPI_COMM_WORLD` appropriately (this can be done, eg, using a simple colouring algorithm).

To support the `PF_WINDOW_RING` mode, the MPI communicator maintains the `first` and `last` integers to keep track of the MPI ranks corresponding to the first and last processors in the current block.

At the beginning of each PFASST iteration the MPI communicator posts receive requests on all PFASST levels.

pthreads

The PFASST pthreads communicator works in `PF_WINDOW_BLOCK` mode but currently only supports one time block. Users are responsible for spawing pthreads and setting up multiple PFASST objects appropriately.

fake

The PFASST fake communicator allows us to study the convergence behaviour of PFASST in `PF_WINDOW_BLOCK` mode using only a single core.

1.1.7 Parameters and variables

The libpfasst library has many parameters which control the behavior of the PFASST algorithm and can be changed by the user. This section lists all the parameters and describes their function, location, and default values. Most of the parameters assume a default value that can be changed by specifying the value either in an input file or on the command line. Some of the parameters must be changed from their default value or PFASST will not execute.

Following these lists is an explanation of how to set parameters through input files or the command line, and how to choose certain parameters to acheive particular variants of PFASST.

Types of parameters

- libpfasst static parameters: are hard coded and cannot be changed at run time
- `pf_pfasst_t` mandatory parameters: must be reassigned at run time the use of default values will result in program termination.
- `pf_pfasst_t` optional parameters: can be reassigned at run time, the use of default values will result in default execution.
- `pf_level_t` mandatory parameters: must be reassigned at run time, the use of default values will result in default execution.
- `pf_level_t` optional parameters: can be reassigned at run time, the use of default values will result in default execution.
- local mandatory parameters: must be passed in a call to `pf_run_pfasst`

- local optional parameters: specified by the user application and unrelated to the workings of libpfasst

libfpasst static parameters

The parameters at the top of the file `src/pf_dtype.f90` are all set at compile time and can't be changed at runtime. The only parameter here of interest to the user is

```
integer, parameter :: pfdp = c_double
```

which controls the precision of all floating point numbers (or at least all those using `pfdp` in the declaration).

pfasst mandatory parameters

The parameters defined in type `pf_pfasst_t` in `src/pf_dtype.f90` are all given a default value. Currently only the variable `nlevels` is given a problematic default. Hence setting this variable on the command line or in an initialization file is mandatory

pfasst optional parameters

The remaining variables in the specification of `pf_pfasst_t` are given default values as below:

```
type :: pf_pfasst_t

   integer :: niters  = 5             ! number of iterations
   integer :: qtype   = SDC_GAUSS_LOBATTO
   integer :: ctype   = SDC_CYCLE_V

   real(pfdp) :: abs_res_tol = 0.d0
   real(pfdp) :: rel_res_tol = 0.d0

   integer :: window = PF_WINDOW_BLOCK

   logical :: Pipeline_G =  .false.
   logical :: PFASST_pred = .false.

   logical :: echo_timings  = .false.
```

These value can be changed as desired.

level mandatory parameters

In the specification of `pf_level_t`, the first two variables `nvars` and `nnodes` are given default values that will cause the program to abort. These variables are typically set per level when initializing PFASST.

```
type :: pf_level_t

   integer     :: nvars = -1          ! number of variables (dofs)
   integer     :: nnodes = -1         ! number of sdc nodes
```

level optional parameters

In the specification of `pf_level_t`, the first variables `nsweeps` and `Finterp` are default values. These can be changed per level as the levels are when initializing PFASST.

```
type :: pf_level_t

   integer     :: nsweeps = 1          ! number of sdc sweeps to perform
   logical     :: Finterp = .false.    ! interpolate functions instead of solutions
```

local mandatory parameters

In the call to run pfasst

```
pf_pfasst_run(pf, q0, dt, tend, nsteps, qend)
```

The variables `q0`, `dt`, and `tend` must be included. The variable `nsteps` is optional, if it is not included, then `nsteps` is set to

```
pf%state%nsteps = ceiling(1.0*tend/dt)
```

Finally, `qend` is also optional and returns the final solution.

> The usual default input file is "probin.nml" wherein the namelist PARAMS (defined locally in probin.f90) can be specified. Alternatively, a different input file can be specified on the command line by adding the file name directly after the executable. The alternative input file must be specified first before any command line parameter specifications (see next section).

File input for pfasst variables

The pfasst parameters are specified in a namelist `PF_PARAMS` defined in routine `pf_read_opts` in `src/pf_options.f90`. This routine is called from `pf_pfasst_create` in `pf_pfasst.f90` (which is typically called when initializing PFASST). If no file is specified in the call to `pf_pfasst_create`, then no file is read. Typically the main routine specifies this input file (the default being probin.nml), and this file can be changed by specifying the value of

> pfasst_nml = 'probin.nml'

either in the local input file or the command line.

Command line input

All the variables in the namelist `PF_PARAMS` can be modified by simply specifying their value on the command line. There is only one caveat to this in that any parameters must be specified after the (optional) input file specification. For example

```
mpirun -n 20 main.exe  myinput.nml niters=10
```

would set the input file to "myinput.nml" and then over-ride any specified value of niters with the value 10. Command line options over-ride input files.

Variables for the predictor

The two variables `Pipeline_G` and `PFASST_pred` determine how the predictor works. The different combinations of these variables and the parameter Nsweeps on the coarsest level great some subtle differences in how the predictor performs.

Some cases:

1. If PFASST_pred is false and Pipeline_G is false, then the predictor is a serial application of SDC with Nsweeps. This can be done without communication wherein every processor mimics the behavior of the processors previous to it in time.
2. If PFASST_pred is false and Pipeline_G is true and Nsweeps is one, then the predictor is a serial application of SDC with 1 sweep. As above, there is no communication necessary.
3. If PFASST_pred is false and Pipeline_G is true and Nsweeps is greater than one, then the predictor is a version of pipelined SDC. There is no communication necessary until the second sweep on the each processor is done. After that, each processor must recieve a new initial value.
4. If PFASST_pred is true, and Nsweeps equals one, then it doesn't matter what Pipeline_G is. No communication is necessary, and we simply reuse the function values from the previous iteration in each SDC sweep. Some care must be taken here as to how to interpret the variable t0 especially in light of time dependent boundary conditions. Currently t0 does not change in these iterations, hence one should use caution using PFASST_pred = true with time dependent boundary conditions.
5. If PFASST_pred is true, and Nsweeps is greater than one and Pipeline_G is true, then the predictor will act like the normal PFASST_pred with Nsweeps equals one, but more iterations will be taken. This choice is a bit strange. No communication is needed until each processor is doing the P+1st iteration, then new initial data must be used and in all cases, previous f values are used in the SDCsweeps. The caveat about t0 is still valid.
6. Finally, if PFASST_pred is true, and Nsweeps is greater than one and Pipeline_G is false, then the predictor will act like the normal PFASST_pred with Nsweeps equals one, but additional iterations are taken before the initial conditions at each processor are reset. This can be done without communication. The caveat about t0 is still valid.

How is this implemented? There are two pieces to the initialization. The first consists of the process of giving every processor an initial value which is consistent with t0 at that processor. This can be done without communication in all cases.