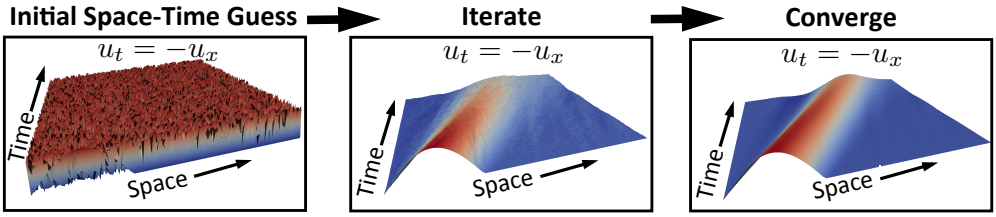
Users' Manual

Version 1.0
August 20, 2014

χBraid

Time-Braid: Multigrid in Time Solvers



V. A. Dobrev, R. D. Falgout, Tz. V. Kolev, N. A. Petersson, J. B. Schroder, U. M. Yang
Center for Applied Scientific Computing (CASC)
Lawrence Livermore National Laboratory

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-******

Contents

1 Abstract

This package implements an optimal-scaling multigrid solver for the linear systems that arise from the discretization of problems with evolutionary behavior. Typically, solution algorithms for evolution equations are based on a time-marching approach, solving sequentially for one time step after the other. Parallelism in these traditional time-integration techniques is limited to spatial parallelism. However, current trends in computer architectures are leading towards systems with more, but not faster, processors. Therefore, faster compute speeds must come from greater parallelism. One approach to achieve parallelism in time is with multigrid, but extending classical multigrid methods for elliptic operators to this setting is a significant achievement. In this software, we implement a non-intrusive, optimal-scaling time-parallel method based on multigrid reduction techniques. The examples in the package demonstrate optimality of our multigrid-reduction-in-time algorithm (MGRIT) for solving a variety of equations in two and three spatial dimensions. These examples can also be used to show that MGRIT can achieve significant speedup in comparison to sequential time marching on modern architectures.

It is **strongly recommended** that you also read `Parallel Time Integration with Multigrid` after reading the Overview of the χBraid Algorithm. It is a more in depth discussion of the algorithm and associated experiments.

2 Introduction

2.1 Meaning of the name

We chose the package name χBraid to stand for *Time-Braid*, where X is the first letter in the Greek work for time, *Chronos*. The algorithm *braids* together time-grids of different granularity in order to create a multigrid method and achieve parallelism in the time dimension. In plain text, we say χBraid, or just Braid for short.

2.2 Overview of the χBraid Algorithm

The goal of χBraid is to solve a problem faster than a traditional time marching algorithm. Instead of sequential time marching, χBraid solves the problem iteratively by simultaneously updating a space-time solution guess over all time values. The initial solution guess can be anything, even a random function over space-time. The iterative updates to the solution guess are done by constructing a hierarchy of temporal grids, where the finest grid contains all of the time values for the simulation. Each subsequent grid is a coarser grid with fewer time values. The coarsest grid has a trivial number of time steps and can be quickly solved exactly. The effect is that solutions to the time marching problem on the coarser (i.e., cheaper) grids can be used to correct the original finest grid solution. Thus, a problem with many time steps (thousands, tens of thousands or more) can be solved with 10 or 15 χBraid iterations, and the overall time to solution can be greatly sped up. However, this is achieved at the cost of more computational resources.

To understand how χBraid differs from traditional time marching, consider the simple linear advection equation, $u_t = -cu_x$. The next figure depicts how one would typically evolve a solution here with sequential time stepping. The initial condition is a wave, and this wave propagates sequentially across space as time increases.

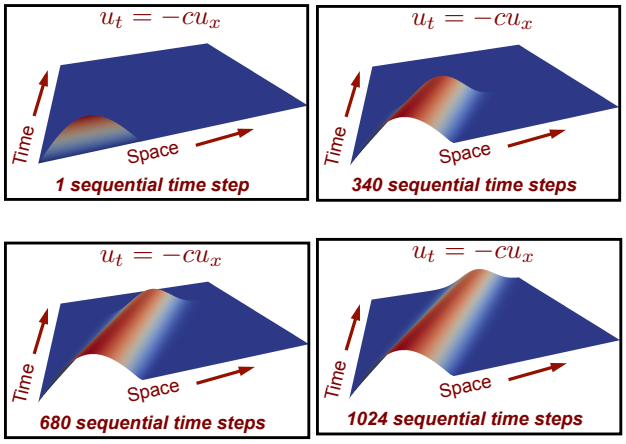


Figure 1: Sequential time stepping.

χBraid instead begins with a solution guess over all of space-time, which for demonstration, we let be random. A χBraid iteration then does

1. Relaxation on the fine grid, i.e., the grid that contains all of the desired time values
   - Relaxation is just a local application of the time stepping scheme, e.g., backward Euler
2. Restriction to the first coarse grid, i.e., interpolate the problem to a grid that contains fewer time values, say every second or every third time value
3. Relaxation on the first coarse grid
4. Restriction to the second coarse grid and so on...
5. When a coarse grid of trivial size (say 2 time steps) is reached, it is solved exactly.
6. The solution is then interpolated from the coarsest grid to the finest grid

One χBraid iteration is called a *cycle* and these cycles continue until the the solution is accurate enough. This is depicted in the next figure, where only a few iterations are required for this simple problem.

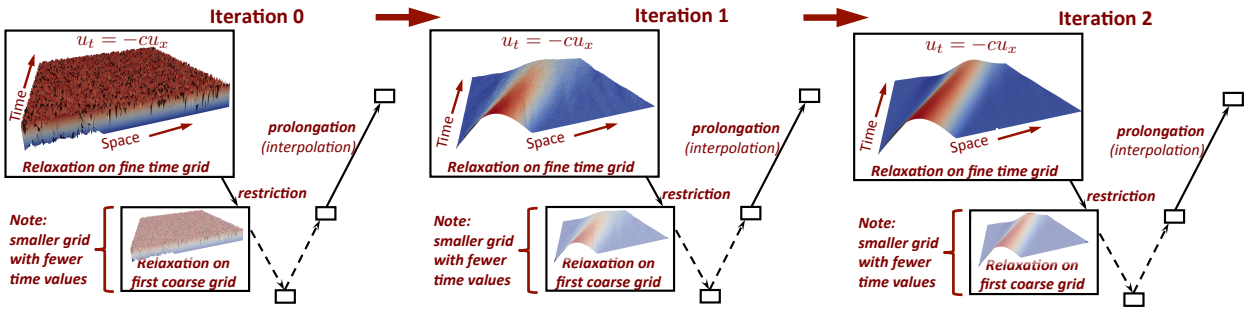


Figure 2: χ Braid iterations.

There are a few important points to make.

- The coarse time grids allow for global propagation of information across space-time with only one χBraid iteration. This is visible in the above figure by observing how the solution is updated from iteration 0 to iteration 1.
- Using coarser (cheaper) grids to correct the fine grid is analagous to spatial multigrid.
- Only a few χBraid iterations are required to find the solution over 1024 time steps. Therefore if enough processors are available to parallelize χBraid, we can see a speedup over traditional time stepping (more on this later).
- This is a simple example, with evenly space time steps. χBraid is structured to handle variable time step sizes and adaptive time step sizes, and these features will be coming.

To firm up our understanding, let's do a little math. Assume that you have a general ODE,

$$u'(t) = f(t,u(t)), \quad u(0) = u_0, \quad t \in [0,T],$$

which you descretize with the one-step integration

$$u_i = \Phi_i(u_{i-1}) + g_i, \quad i = 1,2,...,N.$$

Traditional time marchine would first solve for $i = 1$, then solve for $i = 2$, and so on. This process is equivalent to a forward solve of this system,

$$A\mathbf{u} \equiv \begin{pmatrix} I & & & \\ -\Phi_1 & I & & \\ & \ddots & \ddots & \\ & & -\Phi_N & I \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_N \end{pmatrix} \equiv \mathbf{g}$$

or

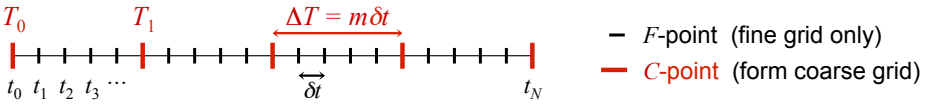
$$A\mathbf{u} = \mathbf{g}.$$

This process is optimal and O(N), but it is sequential. χBraid instead solves the system iteratively, with a multigrid reduction method [1] applied in only the time dimension. This approach is

- nonintrusive, in that it coarsens only in time and the user defines Φ
  - Thus, users can continue using existing time stepping codes by wrapping them into our framework.
- optimal and O(N), but O(N) with a higher constant than time stepping
  - Thus with enough computational resources, χBraid will outperform sequential time stepping.
- highly parallel

χBraid solves this system iteratively by constructing a hierarchy of time grids. We describe the two-grid process, with the multigrid process being a recursive application of the process. We also assume that Φ is constant for notational simplicity.

χBraid functions as follows. The next figure depicts a sample timeline of time values, where the time values have been split into C- and F-points. C-points exist on both the fine and coarse time grid, but F-points exist only on the fine time scale. The first task is relaxation and an effective relaxation alternates between C and F sweeps (this is like

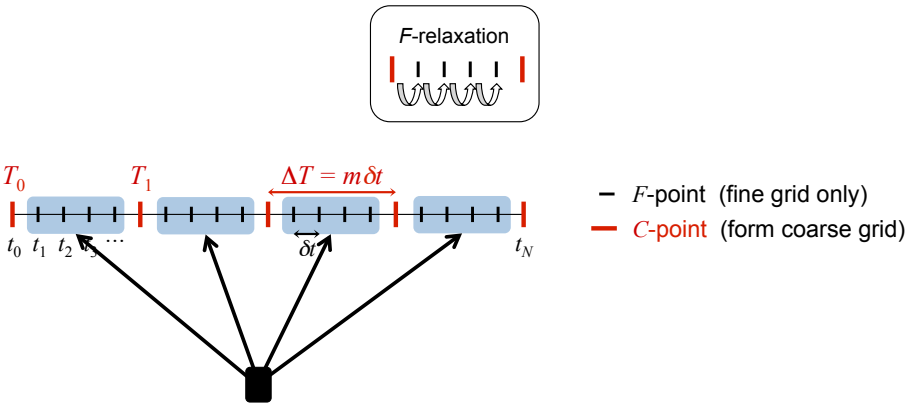


line-relaxation in space in that the residual is set to 0 for an entire time step). An F sweep simply updates time values by integrating with Φ over all the F-points from one C-point to the next, as depicted next.

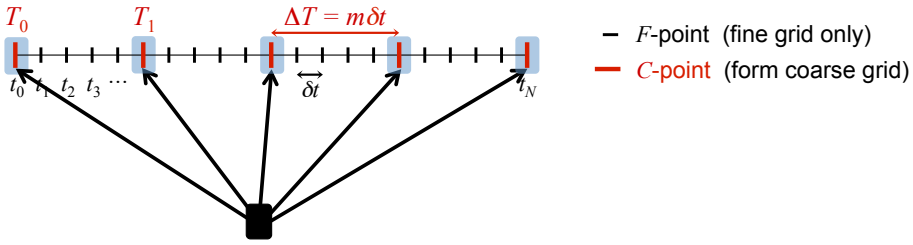
But, such an update can be done simultaneously over all F intervals in parallel, as depicted next.

Following an F sweep we can also do C sweep, as depicted next.

[1] Ries, Manfred, Ulrich Trottenberg, and Gerd Winter. "A note on MGR methods." Linear Algebra and its Applications 49 (1983): 1-26.



- Update all F-points using time propagator Φ



- Update all C-points using time propagator Φ

In general, FCF- and F-relaxation will refer to the relaxation methods used in χBraid. We can say

- FCF or F-relaxtion is highly parallel.
- But, a sequential component exists equaling the the number of F-points between two C-points.
- χBraid uses regular coarsening factors, i.e., the spacing of C-points happens every k points.

After relaxation, comes coarse grid correction. The restriction operator R maps fine grid quantities to the coarse grid by simply injecting values at C-points from the fine grid to the coarse grid,

$$R = \begin{pmatrix} I & & \\ 0 & & \\ \vdots & & \\ 0 & & \\ & I & \\ & 0 & \\ & \vdots & \\ & 0 & \\ & & \ddots \end{pmatrix},$$

where the spacing between each I is $m-1$ block rows. χBraid implements an FAS (Full Approximation Scheme) multigrid cycle, and hence the solution guess and residual (i.e., $A, \mathbf{u}, \mathbf{g} - A\mathbf{u}$) are restricted. This is in contrast to linear multigrid which typically restricts only the residual equation to the coarse grid. We choose FAS because it is *nonlinear* multigrid and allows us to solve nonlinear problems. FAS was invented by Achi Brandt, but this PDF by Van Henson is a good intro.

The main question here is how to form the coarse grid matrix, which in turn asks how to define the coarse grid time

stepper Φ_Δ. It is typical to let Φ_Δ simply be Φ but with the coarse time step size $\Delta T = m\delta t$. Thus if

$$A = \begin{pmatrix} I & & & \\ -\Phi & I & & \\ & \ddots & \ddots & \\ & & -\Phi & I \end{pmatrix}$$

then

$$A_\Delta = \begin{pmatrix} I & & & \\ -\Phi_\Delta & I & & \\ & \ddots & \ddots & \\ & & -\Phi_\Delta & I \end{pmatrix},$$

where A_Δ has fewer rows and columns than A, e.g., if we are coarsening in time by 2, A_Δ has one half as many rows and columns. This coarse grid equation

$$A_\Delta \mathbf{v}_\Delta = \mathbf{g}_\Delta$$

is then solved, where the right-hand-side is defined by FAS (see Two-Grid Algorithm). Finally, FAS defines a coarse grid error approximation $\mathbf{e}_\Delta$, which is interpolated with P_Φ back to the fine grid and added to the current solution guess. Interpolation is equivalent to injecting the coarse grid to the C-points on the fine grid, followed by an F-relaxation sweep. That is,

$$P_\Phi = \begin{pmatrix} I & & \\ \Phi & & \\ \Phi^2 & & \\ \vdots & & \\ \Phi^{m-1} & & \\ & I & \\ & \Phi & \\ & \Phi^2 & \\ & \vdots & \\ & \Phi^{m-1} & \\ & & \ddots \end{pmatrix},$$

where m is the coarsening factor.

2.2.1 Two-Grid Algorithm

This two-grid process is captured with this algorithm. Using a recursive coarse grid solve (i.e., step 3 becomes a recursive call) makes the process multilevel. Halting is done based on a residual tolerance. If the operator is linear, this FAS cycle is equivalent to standard linear multigrid. Note that we represent A as a function below, whereas the above notation was simplified for the linear case.

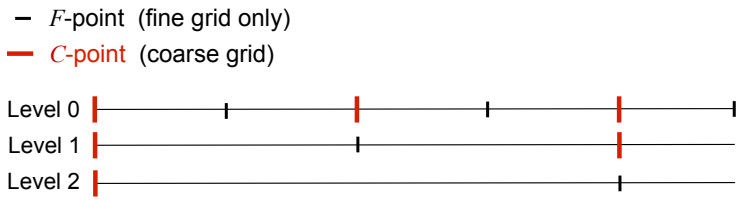
1. Relax on $A(\mathbf{u}) = \mathbf{g}$ using FCF-relaxation
2. Restrict the fine grid approximation and its residual: $\mathbf{u}_\Delta \leftarrow R\mathbf{u}, \quad \mathbf{r}_\Delta \leftarrow R(\mathbf{g} - A(\mathbf{u}))$
3. Solve $A_\Delta(\mathbf{v}_\Delta) = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$
4. Compute the coarse grid error approximation: $\mathbf{e}_\Delta = \mathbf{v}_\Delta - \mathbf{u}_\Delta$
5. Correct: $\mathbf{u} \leftarrow \mathbf{u} + P\mathbf{e}_\Delta$

Caveat: The χBraid implementation of FAS differs slightly from standard FAS. In standard FAS, the error is interpolated to the fine points on the fine grid (here F-points). Instead, given our interpolation operator P_Φ, we add the error to the coarse points on the fine grid (here C-points), and then propagate the *solution* to F-points, like in a reduction method. Thus, F-points are updated in a slightly different, but more exact manner. This strategy allows χBraid to save on storage and to not store F-points, while still effectively solving nonlinear problems.

2.2.2 Summary

In summary, a few points are

- χBraid is an iterative solver for the global space-time problem.
- The user defines the time stepping routine Φ and can wrap existing code to accomplish this.
- χBraid convergence will depend heavily on how well Φ_Δ approximates Φ^m, that is how well a time step size of $m\delta t = \Delta T$ will approximate m applications of the same time integrator for a time step size of δt. This is a subject of research, but this approximation need not capture fine scale behavior, which is instead captured by relaxation on the fine grid.
- The coarsest grid is solved exactly, i.e., sequentially, which can be a bottleneck for two-level methods like Parareal, [2] but not for a multilevel scheme like χBraid where the coarsest grid is of trivial size.
- By forming the coarse grid to have the same sparsity structure and time stepper as the fine grid, the algorithm can recur easily and efficiently.
- Interpolation is ideal or exact, in that an application of interpolation leaves a zero residual at all F-points.
- The process is applied recursively until a trivially sized temporal grid is reached, e.g., 2 or 3 time points. Thus, the coarsening rate m determines how many levels there are in the hierarchy. For instance in this figure, a 3 level hierarchy is shown. Three levels are chosen because there are six time points, $m = 2$ and $m^2 < 6 \leq m^3$. If the coarsening rate had been $m = 4$ then there would only be two levels because, there would be no more points to coarsen!



By default, χBraid will subdivide the time domain into evenly sized time steps. χBraid is structured to handle variable time step sizes and adaptive time step sizes, and these features are coming.

2.3 Overview of the χBraid Code

χBraid is designed to run in conjunction with an existing application code that can be wrapped per our interface. This application code will implement some time marching type simulation like fluid flow. Essentially, the user has to take their application code and extract a stand-alone time-stepping function Φ that can evolve a solution from one time value to another, regardless of time step size. After this is done, the χBraid code takes care of the parallelism in the time dimension.

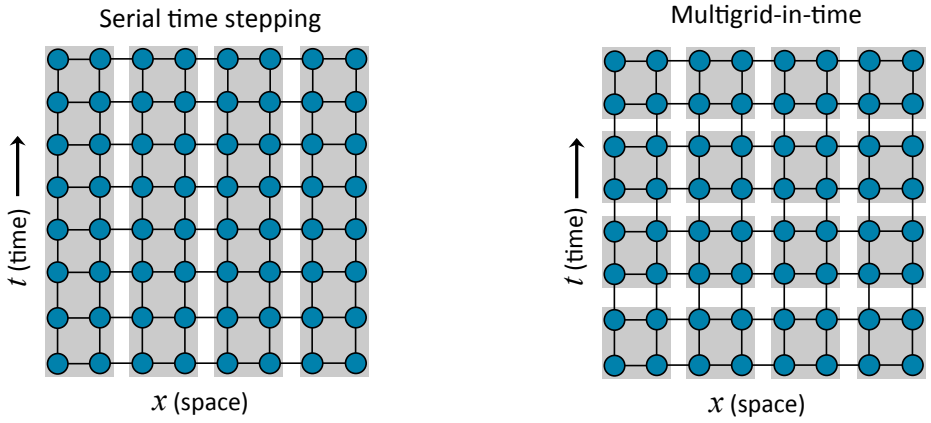
χBraid

- is written in C and can easily interface with Fortran and C++
- uses MPI for parallelism
- self documents through comments in the source code and through $*$.md files
- functions and structures are prefixed by *braid*
  - User routines are prefixed by `braid_`
  - Developer routines are prefixed by `_braid_`

[2] Lions, J., Yvon Maday, and Gabriel Turinici. "A"parareal"in time discretization of PDE's." Comptes Rendus de l'Academie des Sciences Series I Mathematics 332.7 (2001): 661-668.

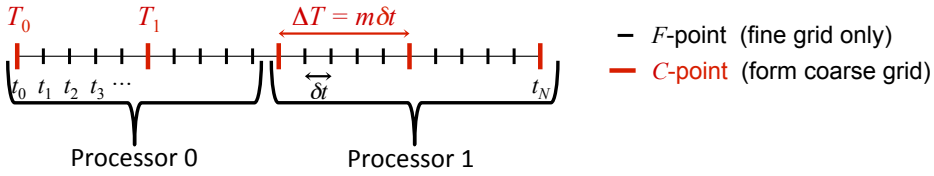
2.3.1 Parallel decomposition and memory

- χBraid decomposes the problem in parallel as depicted next. As you can see, traditional time stepping only



stores one time step at a time, but only enjoys a spatial data decomposition and spatial parallelism. On the other hand, χBraid stores multiple time steps simultaneously and each processor holds a space-time chunk reflecting both the spatial and temporal parallelism.

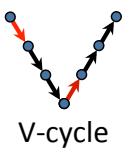
- χBraid only handles temporal parallelism and is agnostic to the spatial decomposition. See braid_SplitCommworld. Each processor owns a certain number of CF intervals of points, as depicted next, where each processor owns 2 CF intervals. χBraid distributes Intervals evenly on the finest grid.



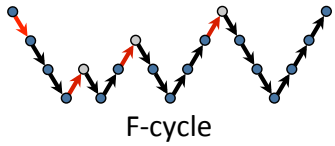
- Storage is greatly minimized by only storing C-points. Whenever an F-point is needed, it is generated by F-relaxation. That is, we only store the red C-point time values in the previous figure. Coarsening can by aggressive with $m = 8, 16, 32$, so the storage requirements of χBraid are significantly reduced when compared to storing all of the time values.
  By only storing data at C-points, we effect a subtle change to the standard FAS algorithm (see Two-Grid Algorithm).
- In practice, storing only one space-time slab is advisable. That is, solve for as many time steps (say k time steps) as you have available memory for. Then move on to the next k time steps.

2.3.2 Cycling and relaxation strategies

There are two main cycling strategies available in χBraid, F-and V-cycles. These two cycles differ in how often and the order in which coarse levels are visited. A V-cycle is depicted next, and is a simple recursive application of the Two-Grid Algorithm.



An F-cycle visits coarse grids more frequently and in a different order. Essentially, an F-cycle uses a V-cycle as the post-smoother, which is an expensive choice for relaxation. But, this extra work gives you a closer approximation to a two-grid cycle, and a faster convergence rate at the extra expense of more work. The effectiveness of a V-cycle as a relaxation scheme can be seen in Figure 2, where one V-cycle globally propagates and *smoothes* the error. The cycling strategy of an F-cycle is depicted next.



Next, we make a few points about F- versus V-cycles.

- One V-cycle iteration is cheaper than one F-cycle iteration.
- But, F-cycles often converge more quickly. For some test cases, this difference can be quite large. The cycle choice for the best time to solution will be problem dependent. See Heat equation example for a case study of cycling strategies.

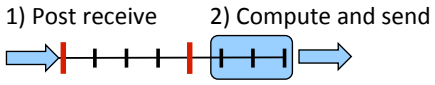
The number of FC relaxation sweeps is another important algorithmic setting. Note that at least one F-relaxation sweep is always done on a level. A few summary points about relaxation are as follows.

- Using FCF (or even FCFCF, FCFCFCF) relaxation, corresponding to passing *braid_SetNRelax* a value of 1, 2 or 3 respectively, will result in an χBraid cycle that converges more quickly as the number of relaxations grows.
- But as the number of relaxations grows, each χBraid cycle becomes more expensive. The optimal relaxation strategy for the best time to solution will be problem dependent.
- However, a good first step is to try FCF on all levels (i.e., *braid_SetNRelax(core, -1, 1)*).
- A common optimization is to first set FCF on all levels (i.e., *braid_setnrelax(core, -1, 1)*), but then overwrite the FCF option on level 0 so that only F-relaxation is done on level 0, (i.e., *braid_setnrelax(core, 0, 1)*). This strategy can work well with F-cycles.
- See Heat equation example for a case study of relaxation strategies.

Last, `Parallel Time Integration with Multigrid` has a more in depth case study of cycling and relaxation strategies

2.3.3 Overlapping communication and computation

χBraid effectively overlaps communication and computation. The main computational kernel of χBraid is relaxation (C or F). At the start a each sweep, each processor first posts a send at its left-most point, and then carries out F-relaxation on its right-most interval in order to send the next processor the data that it needs. If each processor has multiple intervals at this χBraid level, this should allow for complete overlap.



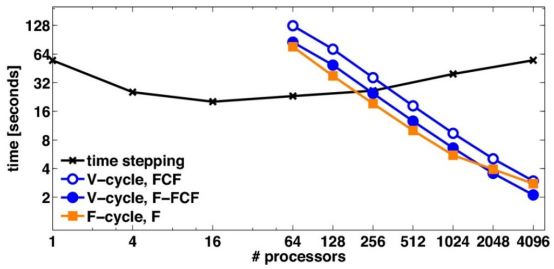
2.3.4 Configuring the χBraid Hierarchy

Some of the more basic χBraid function calls allow you to control aspects discussed here.

- braid_SetFMG: switches between using F- and V-cycles.
- braid_SetMaxIter: sets the maximum number of χBraid iterations
- braid_SetCFactor: sets the coarsening factor for any (or all levels)
- braid_SetNRelax: sets the number of CF-relaxation sweeps for any (or all levels)
- braid_SetRelTol, braid_SetAbsTol: sets the stopping tolerance
- braid_SetMaxCoarse: sets the maximum coarse grid size, in terms of C-points
- braid_SetMaxLevels: sets the maximum number of levels in the χBraid hierarchy

2.3.5 Heat equation example

Here is some experimental data for the 2D heat equation, $u_t = u_{xx} + u_{yy}$ generated by examples/drive-02. The problem



setup is as follows.

- Backwards Euler is used as the time stepper.
- We used a Linux cluster with 4 cores per node, a Sandybridge Intel chipset, and a fast Infiniband interconnect.
- The space-time problem size was $129^2 \times 16,192$ over the unit cube $[0,1] \times [0,1] \times [0,1]$.
- The coarsening factor was $m = 16$ on the finest level and $m = 2$ on coarser levels.
- Since 16 processors optimized the serial time stepping approach, 16 processors in space are also used for the χBraid experiments. So for instance 512 processrs in the plot corresponds to 16 processors in space and 32 processors in time, $16 * 32 = 512$. Thus, each processor owns a space-time hypercube of $(129^2/16) \times (16,192/32)$. See Parallel decomposition and memory for a depiction of how χBraid breaks the problem up.
- Various relaxation and V and F cycling strategies are experimented with.
  - *V-cycle, FCF* denotes V-cycles and FCF-relaxation on each level.
  - *V-cycle, F-FCF* denotes V-cycles and F-relaxation on the finest level and FCF-relaxation on all coarser levels.
  - *F-cycle, F* denotes F-cycles and F-relaxation on each level.
- The initial guess at time values for $t > 0$ is zero, which is typical.

Regarding the performance, we can say

- The best speedup is 10x and this would grow if more processors were available.
- Although not shown, the iteration counts here are about 10-15 χBraid iterations. See `Parallel Time Integration with Multigrid` for the exact iteration counts.
- At smaller core counts, serial time stepping is faster. But at about 256 processors, there is a crossover and χBraid is faster.

- You can see the impact of the cycling and relaxation strategies discussed in Cycling and relaxation strategies. For instance, even though *V-cycle, F-FCF* is a weaker relaxation strategy than *V-cycle, FCF* (i.e., the χBraid convergence is slower), *V-cycle, F-FCF* has a faster time to solution than *V-cycle, FCF* because each cycle is cheaper.
- In general, one level of aggressive coarsening (here by a factor 16) followed by slower coarsening was found to be best on this machine.

Achieving the best speedup can require some tuning, and it is recommended to read Parallel Time Integration with Multigrid where this 2D heat equation example is explored in much more detail.

2.4 Summary

- χBraid applies multigrid to the time dimension.
  - This exposes concurrency in the time dimension.
  - The potential for speedup is large, 10x, 100x, ...
- This is a non-intrusive approach, with an unchanged time discretization defined by user.
- Parallel time integration is only useful beyond some scale. This is evidenced by the experimental results below. For smaller numbers of cores sequential time stepping is faster, but at larger core counts χBraid is much faster.
- The more time steps that you can parallelize over, the better your speedup will be.
- χBraid is optimal for a variety of parabolic problems (see the examples directory).

3 Example

A Simple Example

User Defined Structures and Wrappers

As mentioned, the user must wrap their existing time stepping routine per the χBraid interface. To do this, the user must define two data structures and some wrapper routines. To make the idea more concrete, we now give these function definitions from examples/drive-01, which implements a scalar ODE, $u_t = \lambda u$.

The two data structures are:

1. **App**: This holds a wide variety of information and is *global* in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. Here, this is just the global MPI communicator and few values describing the temporal domain.
```
typedef struct _braid_App_struct
{
   MPI_Comm  comm;
   double    tstart;
   double    tstop;
   int       ntime;

} my_App;
```
2. **Vector**: this defines (roughly) a state vector at a certain time value. It could also contain any other information related to this vector which is needed to evolve the vector to the next time value, like mesh information. Here, the vector is just a scalar double.
```
typedef struct _braid_Vector_struct
{
   double value;

} my_Vector;
```

The user must also define a few wrapper routines. Note, that the app structure is the first argument to every function.

1. **Phi**: This function tells χBraid how to take a time step, and is the core user routine. The user must advance the vector *u* from time *tstart* to time *tstop*. Here advancing the solution just involves the scalar λ. The *rfactor_ptr* and *accuracy* parameters are advanced topics not used here.
   Importantly, the g_i function (from Overview of the χBraid Algorithm) must be incorporated into Phi, so that $\Phi(u_i) \rightarrow u_{i+1}$

```
int
my_Phi(braid_App     app,
       double     tstart,
       double     tstop,
       double     accuracy,
       braid_Vector  u,
       int        *rfactor_ptr)
{
   /* On the finest grid, each value is half the previous value */
   (u->value) = pow(0.5, tstop-tstart)*(u->value);

   /* Zero rhs for now */
   (u->value) += 0.0;

   /* no refinement */
   *rfactor_ptr = 1;

   return 0;
}
```

2. **Init**: This function tells χBraid how to initialize a vector at time *t*. Here that is just allocating and setting a scalar on the heap.

```
int
my_Init(braid_App     app,
        double     t,
        braid_Vector *u_ptr)
{
   my_Vector *u;

   u = (my_Vector *) malloc(sizeof(my_Vector));
   if (t == 0.0)
   {
      /* Initial guess */
      (u->value) = 1.0;
   }
   else
   {
      /* Random between 0 and 1 */
      (u->value) = ((double)rand()) / RAND_MAX;
   }
   *u_ptr = u;

   return 0;
}
```

3. **Clone**: This function tells χBraid how to clone a vector into a new vector.

```
int
my_Clone(braid_App     app,
         braid_Vector  u,
         braid_Vector *v_ptr)
{
   my_Vector *v;

   v = (my_Vector *) malloc(sizeof(my_Vector));
   (v->value) = (u->value);
   *v_ptr = v;

   return 0;
}
```

4. **Free**: This function tells χBraid how to free a vector.

```
int
my_Free(braid_App    app,
        braid_Vector u)
{
   free(u);

   return 0;
}
```

5. **Sum**: This function tells χBraid how to sum two vectors (AXPY operation).

```
int
my_Sum(braid_App    app,
       double    alpha,
       braid_Vector x,
       double    beta,
       braid_Vector y)
{
   (y->value) = alpha*(x->value) + beta*(y->value);

   return 0;
}
```

6. **Dot**: This function tells χBraid how to take the dot product of two vectors.

```
int
my_Dot(braid_App     app,
       braid_Vector  u,
       braid_Vector  v,
       double    *dot_ptr)
{
   double dot;

   dot = (u->value)*(v->value);
   *dot_ptr = dot;

   return 0;
}
```

7. **Write**: This function tells χBraid how to write a vector at time *t* to screen, file, etc... The user defines what is appropriate output. Notice how you are told the time value of the vector *u* and even more information in *status*. This lets you tailor the output to only certain time values.
   If write_level is 2 (see braid_SetWriteLevel), then *Write* is called every χBraid iteration and on every χBraid level. In this case, *status* can be querried using the braid_Get**Status() functions, to determine the current χBraid level and iteration. This allows for even more detailed tracking of the simulation.
   See examples/drive-02 and examples/drive-04 for more advanced uses of the Write function. Drive-04 writes to a GLVIS visualization port, and examples/drive-02 writes to .vtu files.

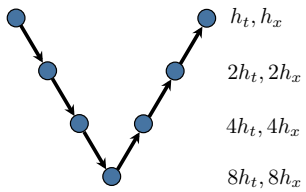
```
int
my_Write(braid_App     app,
         double     t,
         braid_Status  status,
         braid_Vector  u)
{
   MPI_Comm   comm   = (app->comm);
   double     tstart = (app->tstart);
   double     tstop  = (app->tstop);
   int        ntime  = (app->ntime);
   int        index, myid;
   char       filename[255];
   FILE      *file;

   index = ((t-tstart) / ((tstop-tstart)/ntime) + 0.1);

   MPI_Comm_rank(comm, &myid);

   sprintf(filename, "%s.%07d.%05d", "drive-01.out", index, myid);
   file = fopen(filename, "w");
   fprintf(file, "%.14e\n", (u->value));
   fflush(file);
   fclose(file);
```

```
   return 0;
}
```

8. **BufSize**, **BufPack**, **BufUnpack**: These three routines tell χBraid how to communicate vectors between processors. *BufPack* packs a vector into a `void *` buffer for MPI and then *BufUnPack* unpacks it from `void *` to vector. Here doing that for a scalar is trivial. *BufSize* computes the upper bound for the size of an arbitrary vector.

```
int
my_BufSize(braid_App  app,
           int    *size_ptr)
{
   *size_ptr = sizeof(double);
   return 0;
}

int
my_BufPack(braid_App     app,
           braid_Vector  u,
           void      *buffer)
{
   double *dbuffer = buffer;

   dbuffer[0] = (u->value);

   return 0;
}

int
my_BufUnpack(braid_App     app,
             void      *buffer,
             braid_Vector *u_ptr)
{
   double    *dbuffer = buffer;
   my_Vector *u;

   u = (my_Vector *) malloc(sizeof(my_Vector));
   (u->value) = dbuffer[0];
   *u_ptr = u;

   return 0;
}
```

9. **Coarsen**, **Restrict** (optional): These are advanced options that allow for coarsening in space while you coarsen in time. This is useful for maintaining stable explicit schemes on coarse time scales and is not needed here. See for instance examples/drive-04 and examples/drive-05 which use these routines.
   These functions allow you vary the spatial mesh size on χBraid levels as depicted here where the spatial and temporal grid sizes are halved every level.



10. Adaptive and variable time stepping is in the works to be implemented. The *rfactor* parameter in *Phi* will allow this.

Running χBraid

A typical flow of events in the `main` function is to first initialize the `app` structure.

```
/* set up app structure */
app = (my_App *) malloc(sizeof(my_App));
```

```
(app->comm)   = comm;
(app->tstart) = tstart;
(app->tstop)  = tstop;
(app->ntime)  = ntime;
```

Then, the data structure definitions and wrapper routines are passed to χBraid. The core structure is used by χBraid for internal data structures.

```
braid_Core  core;
braid_Init(MPI_COMM_WORLD, comm, tstart, tstop, ntime, app,
        my_Phi, my_Init, my_Clone, my_Free, my_Sum, my_Dot, my_Write,
        my_BufSize, my_BufPack, my_BufUnpack,
        &core);
```

Then, χBraid options are set.

```
braid_SetPrintLevel( core, 1);
braid_SetMaxLevels(core, max_levels);
braid_SetNRelax(core, -1, nrelax);
braid_SetAbsTol(core, tol);
braid_SetCFactor(core, -1, cfactor);
braid_SetMaxIter(core, max_iter);
```

Then, the simulation is run.

```
braid_Drive(core);
```

Then, we clean up.

```
braid_Destroy(core);
```

Finally, to run drive-01, type

```
drive-01 -ml 5
```

This will run drive-01. See examples/drive-0$*$ for more extensive examples.

Testing χBraid

The best overall test for χBraid, is to set the maximum number of levels to 1 (see braid_SetMaxLevels) which will carry out a sequential time stepping test. Take the output given to you by your *Write* function and compare it to output from a non-χBraid run. Is everything OK? Once this is complete, repeat for multilevel χBraid, and check that the solution is correct (that is, it matches a serial run to within tolerance).

At a lower level, to do sanity checks of your data structures and wrapper routines, there are also χBraid test functions, which can be easily run. The test routines also take as arguments the app structure, spatial communicator comm_x, a stream like stdout for test output and a time step size to test dt. After these arguments, function pointers to wrapper routines are the rest of the arguments. Some of the tests can return a boolean variable to indicate correctness.

```
/* Test init(), write(), free() */
braid_TestInitWrite( app, comm_x, stdout, dt, my_Init, my_Write, my_Free);

/* Test clone() */
braid_TestClone( app, comm_x, stdout, dt, my_Init, my_Write, my_Free, my_Clone);

/* Test sum() */
braid_TestSum( app, comm_x, stdout, dt, my_Init, my_Write, my_Free, my_Clone, my_Sum);

/* Test dot() */
correct = braid_TestDot( app, comm_x, stdout, dt, my_Init, my_Free, my_Clone, my_Sum, my_Dot);

/* Test bufsize(), bufpack(), bufunpack() */
```

```
correct = braid_TestBuf( app, comm_x, stdout, dt, my_Init, my_Free, my_Sum, my_Dot, my_BufSize, my_BufPack, my_Bu

/* Test coarsen and refine */
correct = braid_TestCoarsenRefine(app, comm_x, stdout, 0.0, dt, 2*dt, my_Init,
                      my_Write, my_Free, my_Clone, my_Sum, my_Dot, my_CoarsenInjection,
                      my_Refine);
correct = braid_TestCoarsenRefine(app, comm_x, stdout, 0.0, dt, 2*dt, my_Init,
                     my_Write, my_Free, my_Clone, my_Sum, my_Dot, my_CoarsenBilinear,
                     my_Refine);
```

4 Building Braid

- To specify the compilers, flags and options for your machine, edit makefile.inc. For now, we keep it simple and avoid using configure or cmake.
- To make the library, libbraid.a,
  ```
  $ make
  ```
- To make the examples
  ```
  $ make all
  ```
- The makefile lets you pass some parameters like debug with
  ```
  $ make debug=yes
  ```
  or
  ```
  $ make all debug=yes
  ```
  It would also be easy to add additional parameters, e.g., to compile with insure.
- To set compilers and library locations, look in makefile.inc where you can set up an option for your machine to define simple stuff like
  ```
  CC = mpicc
  MPICC = mpicc
  MPICXX = mpiCC
  LFLAGS = -lm
  ```

5 Compiling and running the examples

Type

```
drive-0* -help
```

for instructions on how to run any driver.

To run the examples, type

```
mpirun -np 4 drive-*  [args]
```

1. drive-01 is the simplest example. It implements a scalar ODE and can be compiled and run with no outside dependencies.
2. drive-02 implements the 2D heat equation on a regular grid. You must have `hypre` installed and these variables in examples/Makefile set correctly
   ```
   HYPRE_DIR = ../../linear_solvers/hypre
   HYPRE_FLAGS = -I$(HYPRE_DIR)/include
   HYPRE_LIB = -L$(HYPRE_DIR)/lib -lHYPRE
   ```
3. drive-03 implements the 3D heat equation on a regular grid, and assumes `hypre` is installed just like drive-02.
4. drive-05 implements the 2D heat equation on a regular grid, but it uses spatial coarsening. This allows you to use explicit time stepping on each Braid level, regardless of time step size. It assumes `hypre` is installed just like drive-02.
5. drive-04 is a sophisticated test bed for various PDEs, mostly parabolic. It relies on the `mfem` package to create general finite element discretizations for the spatial problem. Other packages must be installed in this order.

- Unpack and install Metis
- Unpack and install hypre
- Unpack and install mfem. Make the serial version of mfem first by only typing make. Then make sure to set these variables correctly in the mfem Makefile:
```
USE_METIS_5 = YES
HYPRE_DIR  = where_ever_linear_solvers_is/hypre
```
- Make GLVIS, which needs serial mfem. Set these variables in the glvis makefile
```
MFEM_DIR   = mfem_location
MFEM_LIB   = -L$(MFEM_DIR) -lmfem
```
- Go back to the mfem directory and type
```
make clean
make parallel
```
- Go to braid/examples and set these Makefile variables,
```
METIS_DIR = ../../metis-5.1.0/lib
MFEM_DIR = ../../mfem
MFEM_FLAGS = -I$(MFEM_DIR)
MFEM_LIB = -L$(MFEM_DIR) -lmfem -L$(METIS_DIR) -lmetis
```
then type
```
make drive-04
```
- To run drive-04 and glvis, open two windows. In one, start a glvis session
```
./glvis
```
Then, in the other window, run drive-04
```
mpirun -np ... drive-04 [args]
```
Glvis will listen on a port to which drive-04 will dump visualization information.

6 Module Index

6.1 Modules

Here is a list of all modules:

7 File Index

7.1 File List

Here is a list of all files with brief descriptions:

8 Module Documentation

8.1 User-written routines

Typedefs

- typedef struct _braid_App_struct * braid_App
- typedef struct _braid_Vector_struct * braid_Vector
- typedef struct _braid_Status_struct * braid_Status
- typedef braid_Int(* braid_PtFcnPhi)(braid_App app, braid_Real tstart, braid_Real tstop, braid_Real accuracy, braid_Vector u, braid_Int *rfactor_ptr)
- typedef braid_Int(* braid_PtFcnInit)(braid_App app, braid_Real t, braid_Vector *u_ptr)
- typedef braid_Int(* braid_PtFcnClone)(braid_App app, braid_Vector u, braid_Vector *v_ptr)
- typedef braid_Int(* braid_PtFcnFree)(braid_App app, braid_Vector u)
- typedef braid_Int(* braid_PtFcnSum)(braid_App app, braid_Real alpha, braid_Vector x, braid_Real beta, braid_Vector y)
- typedef braid_Int(* braid_PtFcnDot)(braid_App app, braid_Vector u, braid_Vector v, braid_Real *dot_ptr)
- typedef braid_Int(* braid_PtFcnWrite)(braid_App app, braid_Real t, braid_Status status, braid_Vector u)
- typedef braid_Int(* braid_PtFcnBufSize)(braid_App app, braid_Int *size_ptr)
- typedef braid_Int(* braid_PtFcnBufPack)(braid_App app, braid_Vector u, void *buffer)
- typedef braid_Int(* braid_PtFcnBufUnpack)(braid_App app, void *buffer, braid_Vector *u_ptr)
- typedef braid_Int(* braid_PtFcnCoarsen)(braid_App app, braid_Real tstart, braid_Real f_tminus, braid_Real f_tplus, braid_Real c_tminus, braid_Real c_tplus, braid_Vector fu, braid_Vector *cu_ptr)
- typedef braid_Int(* braid_PtFcnRefine)(braid_App app, braid_Real tstart, braid_Real f_tminus, braid_Real f_tplus, braid_Real c_tminus, braid_Real c_tplus, braid_Vector cu, braid_Vector *fu_ptr)

8.1.1 Detailed Description

These are all user-written data structures and routines. There are two data structures (braid_App and braid_Vector) for the user to define. And, there are a variety of function interfaces (defined through function pointer declarations) that the user must implement.

8.1.2 Typedef Documentation

8.1.2.1 typedef struct _braid_App_struct* braid_App

This holds a wide variety of information and is `global` in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. For a simple example, this could just hold the global MPI communicator and a few values describing the temporal domain.

8.1.2.2 typedef braid_Int(* braid_PtFcnBufPack)(braid_App app,braid_Vector u,void *buffer)

This allows Braid to send messages containing braid_Vectors. This routine packs a vector *u* into a *void * buffer* for MPI.

8.1.2.3 typedef braid_Int(* braid_PtFcnBufSize)(braid_App app,braid_Int *size_ptr)

This routine tells Braid message sizes by computing an upper bound in bytes for an arbitrary braid_Vector. This size must be an upper bound for what BufPack and BufUnPack will assume.

8.1.2.4 typedef braid_Int(* braid_PtFcnBufUnpack)(braid_App app,void *buffer,braid_Vector *u_ptr)

This allows Braid to receive messages containing braid_Vectors. This routine unpacks a *void * buffer* from MPI into a braid_Vector.

8.1.2.5 typedef braid_Int(* braid_PtFcnClone)(braid_App app,braid_Vector u,braid_Vector *v_ptr)

Clone *u* into *v_ptr*

8.1.2.6 typedef braid_Int(* braid_PtFcnCoarsen)(braid_App app,braid_Real tstart,braid_Real f_tminus,braid_Real f_tplus,braid_Real c_tminus,braid_Real c_tplus,braid_Vector fu,braid_Vector *cu_ptr)

spatial coarsening (optional). Allows the user to coarsen when going from a fine time grid to a coarse time grid. This function is called on every vector at each level, thus you can coarsem the entire space time domain. The action of this function should match the braid_PtFcnRefine function.

8.1.2.7 typedef braid_Int(* braid_PtFcnDot)(braid_App app,braid_Vector u,braid_Vector v,braid_Real *dot_ptr)

Carry out a dot product *dot_ptr* = <*u*, *v*>

8.1.2.8 typedef braid_Int(* braid_PtFcnFree)(braid_App app,braid_Vector u)

Free and deallocate *u*

8.1.2.9 typedef braid_Int(* braid_PtFcnInit)(braid_App app,braid_Real t,braid_Vector *u_ptr)

Initializes a vector *u_ptr* at time *t*

8.1.2.10 typedef braid_Int(* braid_PtFcnPhi)(braid_App app,braid_Real tstart,braid_Real tstop,braid_Real accuracy,braid_Vector u,braid_Int *rfactor_ptr)

Defines the central time stepping function that the user must write. The user must advance the vector *u* from time *tstart* to time *tstop*. The *rfactor_ptr* and *accuracy* inputs are advanced topics. *rfactor_ptr* allows the user to tell Braid to refine this time interval.

8.1.2.11 typedef braid_Int(* braid_PtFcnRefine)(braid_App app,braid_Real tstart,braid_Real f_tminus,braid_Real f_tplus,braid_Real c_tminus,braid_Real c_tplus,braid_Vector cu,braid_Vector *fu_ptr)

spatial refinement (optional). Allows the user to refine when going from a coarse time grid to a fine time grid. This function is called on every vector at each level, thus you can refine the entire space time domain. The action of this function should match the braid_PtFcnCoarsen function.

8.1.2.12 typedef braid_Int(* braid_PtFcnSum)(braid_App app,braid_Real alpha,braid_Vector x,braid_Real beta,braid_Vector y)

AXPY, *alpha x + beta y –> y*

8.1.2.13 typedef braid_Int(* braid_PtFcnWrite)(braid_App app,braid_Real t,braid_Status status,braid_Vector u)

Write the vector *u* at time *t* to screen, file, etc... The user decides what is appropriate. Notice how you are told the time value of the vector *u* and even more information in *status*. This lets you tailor the output to only certain time values.

If write_level is 2 (see braid_SetWriteLevel), then *write* is called every Braid iteration and on every Braid level. In this case, *status* can be querried using the braid_Get**Status() functions, to determine the current Braid level and iteration. This allows for even more detailed tracking of the simulation.

8.1.2.14 typedef struct _braid_Status_struct* braid_Status

Points to the status structure defined in _braid.h This is NOT a user-defined structure.

8.1.2.15 typedef struct _braid_Vector_struct* braid_Vector

This defines (roughly) a state vector at a certain time value. It could also contain any other information related to this vector which is needed to evolve the vector to the next time value, like mesh information.

8.2 User interface routines

Typedefs

- typedef struct _braid_Core_struct * braid_Core

Functions

- braid_Int braid_Init (MPI_Comm comm_world, MPI_Comm comm, braid_Real tstart, braid_Real tstop, braid_Int ntime, braid_App app, braid_PtFcnPhi phi, braid_PtFcnInit init, braid_PtFcnClone clone, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnDot dot, braid_PtFcnWrite write, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_Core *core_ptr)
- braid_Int braid_Drive (braid_Core core)
- braid_Int braid_Destroy (braid_Core core)
- braid_Int braid_PrintStats (braid_Core core)
- braid_Int braid_SetLoosexTol (braid_Core core, braid_Int level, braid_Real loose_tol)
- braid_Int braid_SetTightxTol (braid_Core core, braid_Int level, braid_Real tight_tol)
- braid_Int braid_SetMaxLevels (braid_Core core, braid_Int max_levels)
- braid_Int braid_SetMaxCoarse (braid_Core core, braid_Int max_coarse)
- braid_Int braid_SetAbsTol (braid_Core core, braid_Real atol)
- braid_Int braid_SetRelTol (braid_Core core, braid_Real rtol)
- braid_Int braid_SetNRelax (braid_Core core, braid_Int level, braid_Int nrelax)
- braid_Int braid_SetCFactor (braid_Core core, braid_Int level, braid_Int cfactor)
- braid_Int braid_SetMaxIter (braid_Core core, braid_Int max_iter)
- braid_Int braid_SetFMG (braid_Core core)
- braid_Int braid_SetNFMGVcyc (braid_Core core, braid_Int nfmg_Vcyc)
- braid_Int braid_SetSpatialCoarsen (braid_Core core, braid_PtFcnCoarsen coarsen)
- braid_Int braid_SetSpatialRefine (braid_Core core, braid_PtFcnRefine refine)
- braid_Int braid_SetPrintLevel (braid_Core core, braid_Int print_level)
- braid_Int braid_SetPrintFile (braid_Core core, const char *printfile_name)
- braid_Int braid_SetWriteLevel (braid_Core core, braid_Int write_level)
- braid_Int braid_SplitCommworld (const MPI_Comm *comm_world, braid_Int px, MPI_Comm *comm_x, MPI_Comm *comm_t)
- braid_Int braid_GetStatusResidual (braid_Status status, braid_Real *rnorm_ptr)
- braid_Int braid_GetStatusIter (braid_Status status, braid_Int *iter_ptr)
- braid_Int braid_GetStatusLevel (braid_Status status, braid_Int *level_ptr)
- braid_Int braid_GetStatusDone (braid_Status status, braid_Int *done_ptr)
- braid_Int braid_GetNumIter (braid_Core core, braid_Int *niter_ptr)
- braid_Int braid_GetRNorm (braid_Core core, braid_Real *rnorm_ptr)

8.2.1 Detailed Description

these are interface routines to initialize and run Braid

8.2.2 Typedef Documentation

8.2.2.1 typedef struct _braid_Core_struct* braid_Core

points to the core structure defined in _braid.h

8.2.3 Function Documentation

8.2.3.1 braid_Int braid_Destroy (braid_Core *core*)

Clean up and destroy core.

Parameters

core	braid_Core (_braid_Core) struct

8.2.3.2 braid_Int braid_Drive (braid_Core *core*)

Carry out a simulation with Braid. Integrate in time.

Parameters

core	braid_Core (_braid_Core) struct

8.2.3.3 braid_Int braid_GetNumIter (braid_Core *core,* braid_Int * *niter_ptr*)

After Drive() finishes, this returns the number of iterations taken.

Parameters

core	braid_Core (_braid_Core) struct
niter_ptr	output, holds number of iterations taken

8.2.3.4 braid_Int braid_GetRNorm (braid_Core *core,* braid_Real * *rnorm_ptr*)

After Drive() finishes, this returns the last measured residual norm.

Parameters

core	braid_Core (_braid_Core) struct
rnorm_ptr	output, holds final residual norm

8.2.3.5 braid_Int braid_GetStatusDone (braid_Status *status,* braid_Int * *done_ptr*)

Return whether Braid is done for the current status object

done_ptr = 1 indicates that this is the last call to Write, because Braid has stopped iterating (either maxiter has been reached, or the tolerance has been met).

Parameters

status	structure containing current simulation info
done_ptr	output, =1 if Braid has finished and this is the final Write, else =0

8.2.3.6 braid_Int braid_GetStatusIter (braid_Status *status,* braid_Int * *iter_ptr*)

Return the iteration for the current status object.

Parameters

status	structure containing current simulation info
iter_ptr	output, current iteration number

8.2.3.7 braid_Int braid_GetStatusLevel (braid_Status *status,* braid_Int * *level_ptr*)

Return the Braid level for the current status object.

Parameters

status	structure containing current simulation info
level_ptr	output, current level in Braid

8.2.3.8 braid_Int braid_GetStatusResidual (braid_Status *status,* braid_Real * *rnorm_ptr*)

Return the residual for the current status object.

Parameters

status	structure containing current simulation info
rnorm_ptr	output, current residual norm

8.2.3.9 braid_Int braid_Init (MPI_Comm *comm_world,* MPI_Comm *comm,* braid_Real *tstart,* braid_Real *tstop,* braid_Int *ntime,* braid_App *app,* braid_PtFcnPhi *phi,* braid_PtFcnInit *init,* braid_PtFcnClone *clone,* braid_PtFcnFree *free,* braid_PtFcnSum *sum,* braid_PtFcnDot *dot,* braid_PtFcnWrite *write,* braid_PtFcnBufSize *bufsize,* braid_PtFcnBufPack *bufpack,* braid_PtFcnBufUnpack *bufunpack,* braid_Core * *core_ptr*)

Create a core object with the required initial data.

This core is used by Braid for internal data structures. The output is *core_ptr* which points to the newly created braid_-Core structure.

Parameters

comm_world	Global communicator for space and time
comm	Communicator for temporal dimension
tstart	start time
tstop	End time
ntime	Initial number of temporal grid values
app	User-defined _braid_App structure
phi	User time stepping routine to advance a braid_Vector forward one step
init	Initialize a braid_Vector on the finest temporal grid
clone	Clone a braid_Vector
free	Free a braid_Vector
sum	Compute vector sum of two braid_Vectors
dot	Compute dot product between two braid_Vectors
write	Writes a braid_Vector to file, screen
bufsize	Computes size for MPI buffer for one braid_Vector
bufpack	Packs MPI buffer to contain one braid_Vector
bufunpack	Unpacks MPI buffer into a braid_Vector
core_ptr	Pointer to braid_Core (_braid_Core) struct

8.2.3.10 braid_Int braid_PrintStats (braid_Core *core*)

Print statistics after a Braid run.

Parameters

core	braid_Core (_braid_Core) struct

8.2.3.11 braid_Int braid_SetAbsTol (braid_Core *core,* braid_Real *atol*)

Set absolute stopping tolerance.

Recommended option over relative tolerance

Parameters

core	braid_Core (_braid_Core) struct
atol	absolute stopping tolerance

8.2.3.12 braid_Int braid_SetCFactor (braid_Core *core,* braid_Int *level,* braid_Int *cfactor*)

Set the coarsening factor *cfactor* on grid *level* (level 0 is the finest grid). The default factor is 2 on all levels. To change the default factor, use *level = -1*.

Parameters

core	braid_Core (_braid_Core) struct
level	*level* to set coarsening factor on
cfactor	desired coarsening factor

8.2.3.13 braid_Int braid_SetFMG (braid_Core *core*)

Once called, Braid will use FMG (i.e., F-cycles.

Parameters

core	braid_Core (_braid_Core) struct

8.2.3.14 braid_Int braid_SetLoosexTol (braid_Core *core,* braid_Int *level,* braid_Real *loose_tol*)

Set loose stopping tolerance *loose_tol* for spatial solves on grid *level* (level 0 is the finest grid).

Parameters

core	braid_Core (_braid_Core) struct
level	level to set *loose_tol*
loose_tol	tolerance to set

8.2.3.15 braid_Int braid_SetMaxCoarse (braid_Core *core,* braid_Int *max_coarse*)

Set max allowed coarse grid size (in terms of C-points)

Parameters

core	braid_Core (_braid_Core) struct
max_coarse	maximum coarse grid size

8.2.3.16 braid_Int braid_SetMaxIter (braid_Core *core,* braid_Int *max_iter*)

Set max number of multigrid iterations.

Parameters

core	braid_Core (_braid_Core) struct
max_iter	maximum iterations to allow

8.2.3.17 braid_Int braid_SetMaxLevels (braid_Core *core,* braid_Int *max_levels*)

Set max number of multigrid levels.

Parameters

core	braid_Core (_braid_Core) struct
max_levels	maximum levels to allow

8.2.3.18 braid_Int braid_SetNFMGVcyc (braid_Core *core*, braid_Int *nfmg_Vcyc*)

Set number of V-cycles to use at each FMG level (standard is 1)

Parameters

core	braid_Core (_braid_Core) struct
nfmg_Vcyc	number of V-cycles to do each FMG level

8.2.3.19 braid_Int braid_SetNRelax (braid_Core *core*, braid_Int *level*, braid_Int *nrelax*)

Set the number of relaxation sweeps *nrelax* on grid *level* (level 0 is the finest grid). The default is 1 on all levels. To change the default factor, use *level = -1*. One sweep is a CF relaxation sweep.

Parameters

core	braid_Core (_braid_Core) struct
level	*level* to set *nrelax* on
nrelax	number of relaxations to do on *level*

8.2.3.20 braid_Int braid_SetPrintFile (braid_Core *core*, const char * *printfile_name*)

Set output file for runtime print messages. Level of printing is controlled by braid_SetPrintLevel. Default is stdout.

Parameters

core	braid_Core (_braid_Core) struct
printfile_name	output file for Braid runtime output

8.2.3.21 braid_Int braid_SetPrintLevel (braid_Core *core*, braid_Int *print_level*)

Set print level for Braid. This controls how much information is printed to the Braid print file (braid_SetPrintFile).

- Level 0: no output
- Level 1: print typical information like a residual history, number of levels in the Braid hierarchy, and so on.
- Level 2: level 1 output, plus debug level output.

Default is level 1.

Parameters

core	braid_Core (_braid_Core) struct
print_level	desired print level

8.2.3.22 braid_Int braid_SetRelTol (braid_Core *core*, braid_Real *rtol*)

Set relative stopping tolerance, relative to the initial residual. Be careful. If your initial guess is all zero, then the initial residual may only be nonzero over one or two time values, and this will skew the relative tolerance. Absolute tolerances are recommended.

Parameters

core	braid_Core (_braid_Core) struct
rtol	relative stopping tolerance

8.2.3.23 braid_Int braid_SetSpatialCoarsen (braid_Core *core,* braid_PtFcnCoarsen *coarsen*)

Set spatial coarsening routine with user-defined routine. Default is no spatial refinment or coarsening.

Parameters

core	braid_Core (_braid_Core) struct
coarsen	function pointer to spatial coarsening routine

8.2.3.24 braid_Int braid_SetSpatialRefine (braid_Core *core,* braid_PtFcnRefine *refine*)

Set spatial refinement routine with user-defined routine. Default is no spatial refinment or coarsening.

Parameters

core	braid_Core (_braid_Core) struct
refine	function pointer to spatial refinement routine

8.2.3.25 braid_Int braid_SetTightxTol (braid_Core *core,* braid_Int *level,* braid_Real *tight_tol*)

Set tight stopping tolerance *tight_tol* for spatial solves on grid *level* (level 0 is the finest grid).

Parameters

core	braid_Core (_braid_Core) struct
level	level to set *tight_tol*
tight_tol	tolerance to set

8.2.3.26 braid_Int braid_SetWriteLevel (braid_Core *core,* braid_Int *write_level*)

Set write level for Braid. This controls how often the user's write routine is called.

- Level 0: Never call the user's write routine
- Level 1: Only call the user's write routine after Braid is finished
- Level 2: Call the user's write routine every iteration and on every level. This is during _braid_FRestrict, during the down-cycle part of a Braid iteration.

Default is level 1.

Parameters

core	braid_Core (_braid_Core) struct
write_level	desired write_level

8.2.3.27 braid_Int braid_SplitCommworld (const MPI_Comm * *comm_world,* braid_Int *px,* MPI_Comm * *comm_x,* MPI_Comm * *comm_t*)

Split MPI commworld into *comm_x* and *comm_t*, the spatial and temporal communicators. The total number of processors will equal Px∗Pt, there Px is the number of procs in space, and Pt is the number of procs in time.

Parameters

comm_world	Global communicator to split
px	Number of processors parallelizing space for a single time step
comm_x	Spatial communicator (written as output)
comm_t	Temporal communicator (written as output)

8.3 Braid test routines

Functions

- braid_Int braid_TestInitWrite (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnWrite write, braid_PtFcnFree free)
- braid_Int braid_TestClone (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnWrite write, braid_PtFcnFree free, braid_PtFcnClone clone)
- braid_Int braid_TestSum (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnWrite write, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum)
- braid_Int braid_TestDot (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnDot dot)
- braid_Int braid_TestBuf (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnDot dot, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack)
- braid_Int braid_TestCoarsenRefine (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnWrite write, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnDot dot, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)
- braid_Int braid_TestAll (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnDot dot, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)

8.3.1 Detailed Description

These are sanity check routines to help a user test their Braid code.

8.3.2 Function Documentation

8.3.2.1 braid_Int braid_TestAll (braid_App *app*, MPI_Comm *comm_x*, FILE * *fp*, braid_Real *t*, braid_Real *fdt*, braid_Real *cdt*, braid_PtFcnInit *init*, braid_PtFcnFree *free*, braid_PtFcnClone *clone*, braid_PtFcnSum *sum*, braid_PtFcnDot *dot*, braid_PtFcnBufSize *bufsize*, braid_PtFcnBufPack *bufpack*, braid_PtFcnBufUnpack *bufunpack*, braid_PtFcnCoarsen *coarsen*, braid_PtFcnRefine *refine*)

Runs all of the individual braid_Test* routines

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to initialize test vectors with

fdt	Fine time step value that you spatially coarsen from
cdt	Coarse time step value that you coarsen to
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
dot	Compute dot product of two braid_Vectors
bufsize	Computes size in bytes for one braid_Vector MPI buffer
bufpack	Packs MPI buffer to contain one braid_Vector
bufunpack	Unpacks MPI buffer into a braid_Vector
coarsen	Spatially coarsen a vector. If NULL, test is skipped.
refine	Spatially refine a vector. If NULL, test is skipped.

8.3.2.2 braid_Int braid_TestBuf (braid_App *app,* MPI_Comm *comm_x,* FILE * *fp,* braid_Real *t,* braid_PtFcnInit *init,* braid_PtFcnFree *free,* braid_PtFcnSum *sum,* braid_PtFcnDot *dot,* braid_PtFcnBufSize *bufsize,* braid_PtFcnBufPack *bufpack,* braid_PtFcnBufUnpack *bufunpack*)

Test the BufPack, BufUnpack and BufSize functions.

A vector is initialized at time *t*, packed into a buffer, then unpacked from a buffer. The unpacked result must equal the original vector.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test Buffer routines (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
sum	Compute vector sum of two braid_Vectors
dot	Compute dot product of two braid_Vectors
bufsize	Computes size in bytes for one braid_Vector MPI buffer
bufpack	Packs MPI buffer to contain one braid_Vector
bufunpack	Unpacks MPI buffer containing one braid_Vector

8.3.2.3 braid_Int braid_TestClone (braid_App *app,* MPI_Comm *comm_x,* FILE * *fp,* braid_Real *t,* braid_PtFcnInit *init,* braid_PtFcnWrite *write,* braid_PtFcnFree *free,* braid_PtFcnClone *clone*)

Test the clone function.

A vector is initialized at time *t*, cloned, and both vectors are written. Then both vectors are free-d. The user is to check (via the write function) to see if it is identical.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test clone with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
write	Write a braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector

8.3.2.4 braid_Int braid_TestCoarsenRefine (braid_App *app*, MPI_Comm *comm_x*, FILE * *fp*, braid_Real *t*, braid_Real *fdt*, braid_Real *cdt*, braid_PtFcnInit *init*, braid_PtFcnWrite *write*, braid_PtFcnFree *free*, braid_PtFcnClone *clone*, braid_PtFcnSum *sum*, braid_PtFcnDot *dot*, braid_PtFcnCoarsen *coarsen*, braid_PtFcnRefine *refine*)

Test the Coarsen and Refine functions.

A vector is initialized at time *t*, and various sanity checks on the spatial coarsening and refinement routines are run.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to initialize test vectors
fdt	Fine time step value that you spatially coarsen from
cdt	Coarse time step value that you coarsen to
init	Initialize a braid_Vector on finest temporal grid
write	Write a braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
dot	Compute dot product of two braid_Vectors
coarsen	Spatially coarsen a vector
refine	Spatially refine a vector

8.3.2.5 braid_Int braid_TestDot (braid_App *app*, MPI_Comm *comm_x*, FILE * *fp*, braid_Real *t*, braid_PtFcnInit *init*, braid_PtFcnFree *free*, braid_PtFcnClone *clone*, braid_PtFcnSum *sum*, braid_PtFcnDot *dot*)

Test the dot function.

A vector is initialized at time *t* and then cloned. Various dot products like <3 v, v>/<v, v> are computed with known output, e.g., <3 v, v>/<v, v> must equal 3. If all the tests pass, then 1 is returned.

- Returns 0 if the tests fail
- Returns 1 if the tests pass
- Check the log messages to see details of which tests failed.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test Dot with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors
dot	Compute dot product of two braid_Vectors

8.3.2.6 braid_Int braid_TestInitWrite (braid_App *app*, MPI_Comm *comm_x*, FILE * *fp*, braid_Real *t*, braid_PtFcnInit *init*, braid_PtFcnWrite *write*, braid_PtFcnFree *free*)

Test the init, write and free functions.

A vector is initialized at time *t*, written, and then free-d

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test init with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
write	Write a braid_Vector (can be NULL for no writing)
free	Free a braid_Vector

8.3.2.7 braid_Int braid_TestSum (braid_App *app*, MPI_Comm *comm_x*, FILE * *fp*, braid_Real *t*, braid_PtFcnInit *init*, braid_PtFcnWrite *write*, braid_PtFcnFree *free*, braid_PtFcnClone *clone*, braid_PtFcnSum *sum*)

Test the sum function.

A vector is initialized at time *t*, cloned, and then these two vectors are summed a few times, with the results written. The vectors are then free-d. The user is to check (via the write function) that the output matches the sum of the two original vectors.

Parameters

app	User defined App structure
comm_x	Spatial communicator
fp	File pointer (could be stdout or stderr) for log messages
t	Time value to test Sum with (used to initialize the vectors)
init	Initialize a braid_Vector on finest temporal grid
write	Write a braid_Vector (can be NULL for no writing)
free	Free a braid_Vector
clone	Clone a braid_Vector
sum	Compute vector sum of two braid_Vectors

9 File Documentation

9.1 braid.h File Reference

Typedefs

- typedef int braid_Int
- typedef double braid_Real
- typedef struct _braid_App_struct * braid_App
- typedef struct _braid_Vector_struct * braid_Vector
- typedef struct _braid_Status_struct * braid_Status
- typedef braid_Int(* braid_PtFcnPhi)(braid_App app, braid_Real tstart, braid_Real tstop, braid_Real accuracy, braid_Vector u, braid_Int *rfactor_ptr)
- typedef braid_Int(* braid_PtFcnInit)(braid_App app, braid_Real t, braid_Vector *u_ptr)
- typedef braid_Int(* braid_PtFcnClone)(braid_App app, braid_Vector u, braid_Vector *v_ptr)
- typedef braid_Int(* braid_PtFcnFree)(braid_App app, braid_Vector u)
- typedef braid_Int(* braid_PtFcnSum)(braid_App app, braid_Real alpha, braid_Vector x, braid_Real beta, braid_Vector y)
- typedef braid_Int(* braid_PtFcnDot)(braid_App app, braid_Vector u, braid_Vector v, braid_Real *dot_ptr)
- typedef braid_Int(* braid_PtFcnWrite)(braid_App app, braid_Real t, braid_Status status, braid_Vector u)
- typedef braid_Int(* braid_PtFcnBufSize)(braid_App app, braid_Int *size_ptr)
- typedef braid_Int(* braid_PtFcnBufPack)(braid_App app, braid_Vector u, void *buffer)
- typedef braid_Int(* braid_PtFcnBufUnpack)(braid_App app, void *buffer, braid_Vector *u_ptr)
- typedef braid_Int(* braid_PtFcnCoarsen)(braid_App app, braid_Real tstart, braid_Real f_tminus, braid_Real f_tplus, braid_Real c_tminus, braid_Real c_tplus, braid_Vector fu, braid_Vector *cu_ptr)
- typedef braid_Int(* braid_PtFcnRefine)(braid_App app, braid_Real tstart, braid_Real f_tminus, braid_Real f_tplus, braid_Real c_tminus, braid_Real c_tplus, braid_Vector cu, braid_Vector *fu_ptr)
- typedef struct _braid_Core_struct * braid_Core

Functions

- braid_Int braid_Init (MPI_Comm comm_world, MPI_Comm comm, braid_Real tstart, braid_Real tstop, braid_Int ntime, braid_App app, braid_PtFcnPhi phi, braid_PtFcnInit init, braid_PtFcnClone clone, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnDot dot, braid_PtFcnWrite write, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_Core *core_ptr)
- braid_Int braid_Drive (braid_Core core)
- braid_Int braid_Destroy (braid_Core core)
- braid_Int braid_PrintStats (braid_Core core)
- braid_Int braid_SetLoosexTol (braid_Core core, braid_Int level, braid_Real loose_tol)
- braid_Int braid_SetTightxTol (braid_Core core, braid_Int level, braid_Real tight_tol)
- braid_Int braid_SetMaxLevels (braid_Core core, braid_Int max_levels)
- braid_Int braid_SetMaxCoarse (braid_Core core, braid_Int max_coarse)
- braid_Int braid_SetAbsTol (braid_Core core, braid_Real atol)
- braid_Int braid_SetRelTol (braid_Core core, braid_Real rtol)
- braid_Int braid_SetNRelax (braid_Core core, braid_Int level, braid_Int nrelax)
- braid_Int braid_SetCFactor (braid_Core core, braid_Int level, braid_Int cfactor)
- braid_Int braid_SetMaxIter (braid_Core core, braid_Int max_iter)
- braid_Int braid_SetFMG (braid_Core core)
- braid_Int braid_SetNFMGVcyc (braid_Core core, braid_Int nfmg_Vcyc)

- braid_Int braid_SetSpatialCoarsen (braid_Core core, braid_PtFcnCoarsen coarsen)
- braid_Int braid_SetSpatialRefine (braid_Core core, braid_PtFcnRefine refine)
- braid_Int braid_SetPrintLevel (braid_Core core, braid_Int print_level)
- braid_Int braid_SetPrintFile (braid_Core core, const char *printfile_name)
- braid_Int braid_SetWriteLevel (braid_Core core, braid_Int write_level)
- braid_Int braid_SplitCommworld (const MPI_Comm *comm_world, braid_Int px, MPI_Comm *comm_x, MPI_Comm *comm_t)
- braid_Int braid_GetStatusResidual (braid_Status status, braid_Real *rnorm_ptr)
- braid_Int braid_GetStatusIter (braid_Status status, braid_Int *iter_ptr)
- braid_Int braid_GetStatusLevel (braid_Status status, braid_Int *level_ptr)
- braid_Int braid_GetStatusDone (braid_Status status, braid_Int *done_ptr)
- braid_Int braid_GetNumIter (braid_Core core, braid_Int *niter_ptr)
- braid_Int braid_GetRNorm (braid_Core core, braid_Real *rnorm_ptr)

9.1.1 Detailed Description

Define headers for user interface routines. This file contains routines used to allow the user to initialize, run and get and set a Braid solver.

9.1.2 Typedef Documentation

9.1.2.1 typedef int braid_Int

Defines integer type

9.1.2.2 typedef double braid_Real

Defines floating point type

9.2 braid_test.h File Reference

Functions

- braid_Int braid_TestInitWrite (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnWrite write, braid_PtFcnFree free)
- braid_Int braid_TestClone (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnWrite write, braid_PtFcnFree free, braid_PtFcnClone clone)
- braid_Int braid_TestSum (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnWrite write, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum)
- braid_Int braid_TestDot (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnDot dot)
- braid_Int braid_TestBuf (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnSum sum, braid_PtFcnDot dot, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack)
- braid_Int braid_TestCoarsenRefine (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnWrite write, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnDot dot, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)
- braid_Int braid_TestAll (braid_App app, MPI_Comm comm_x, FILE *fp, braid_Real t, braid_Real fdt, braid_Real cdt, braid_PtFcnInit init, braid_PtFcnFree free, braid_PtFcnClone clone, braid_PtFcnSum sum, braid_PtFcnDot dot, braid_PtFcnBufSize bufsize, braid_PtFcnBufPack bufpack, braid_PtFcnBufUnpack bufunpack, braid_PtFcnCoarsen coarsen, braid_PtFcnRefine refine)

9.2.1 Detailed Description

Define headers for Braid test routines. This file contains routines used to test a user's Braid wrapper routines one-by-one.

Index