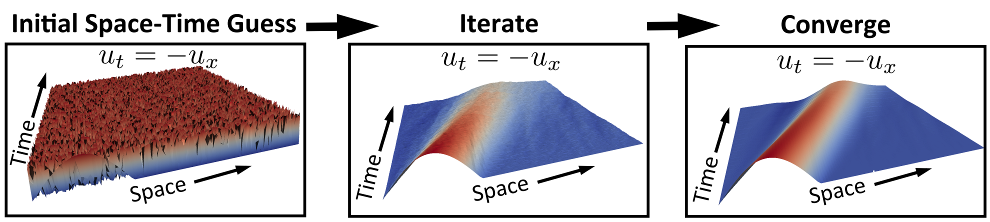
Developers' Manual

Version 1.0
August 18, 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 2

2 Introduction 2

2.1 Meaning of the name 2

2.2 Overview of the Algorithm 2

2.2.1 Two-Grid Algorithm 6

2.2.2 Summary 6

2.3 Overview of the Code 7

2.3.1 Parallel decomposition and memory 7

2.3.2 Cycling and relaxation strategies 8

2.3.3 Overlapping communication and computation 9

2.3.4 Configuring the \form#0Braid Hierarchy 9

2.3.5 Heat equation example 10

2.4 Summary 11

3 Example 11

4 Building Braid 16

5 Compiling and running the examples 16

6 Coding Style 17

7 Using Doxygen 17

8 Regression Testing 18

9 Module Index 22

9.1 Modules 22

10 Data Structure Index 22

10.1 Data Structures 22

11 File Index 23

11.1 File List 23

12 Module Documentation 23

12.1 User-written routines 23

12.1.1 Detailed Description 24

12.1.2 Typedef Documentation 24

12.2 User interface routines 26
12.2.1 Detailed Description 26
12.2.2 Typedef Documentation 26
12.2.3 Function Documentation 27
12.3 Braid test routines 35
12.3.1 Detailed Description 35
12.3.2 Function Documentation 35

13 Data Structure Documentation 39
13.1 _braid_AccuracyHandle Struct Reference 39
13.1.1 Detailed Description 39
13.1.2 Field Documentation 39
13.2 _braid_CommHandle Struct Reference 39
13.2.1 Detailed Description 40
13.2.2 Field Documentation 40
13.3 _braid_Core Struct Reference 40
13.3.1 Detailed Description 41
13.3.2 Field Documentation 41
13.4 _braid_Grid Struct Reference 44
13.4.1 Detailed Description 45
13.4.2 Field Documentation 45
13.5 _braid_Status Struct Reference 46
13.5.1 Detailed Description 46
13.5.2 Field Documentation 46

14 File Documentation 47
14.1 _braid.h File Reference 47
14.1.1 Detailed Description 48
14.1.2 Macro Definition Documentation 48
14.1.3 Function Documentation 49
14.1.4 Variable Documentation 53
14.2 braid.h File Reference 53
14.2.1 Detailed Description 54
14.2.2 Typedef Documentation 54
14.3 braid_test.h File Reference 55
14.3.1 Detailed Description 55
14.4 util.h File Reference 55
14.4.1 Detailed Description 55

14.4.2 Function Documentation 55

Index **57**

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 intro. 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 χ 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 XBraid, or just Braid for short.

2.2 Overview of the 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.

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

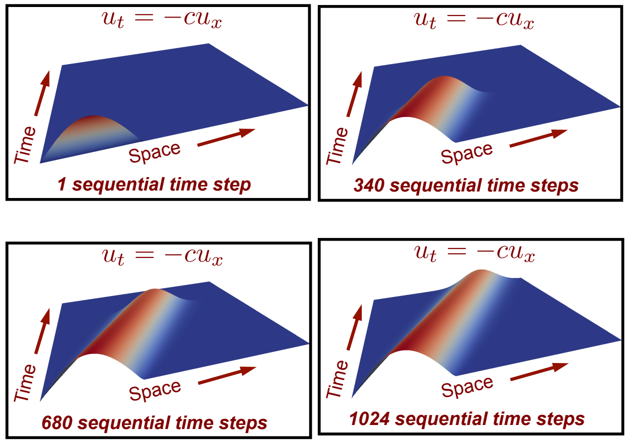


Figure 1: Sequential time stepping.

- 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.

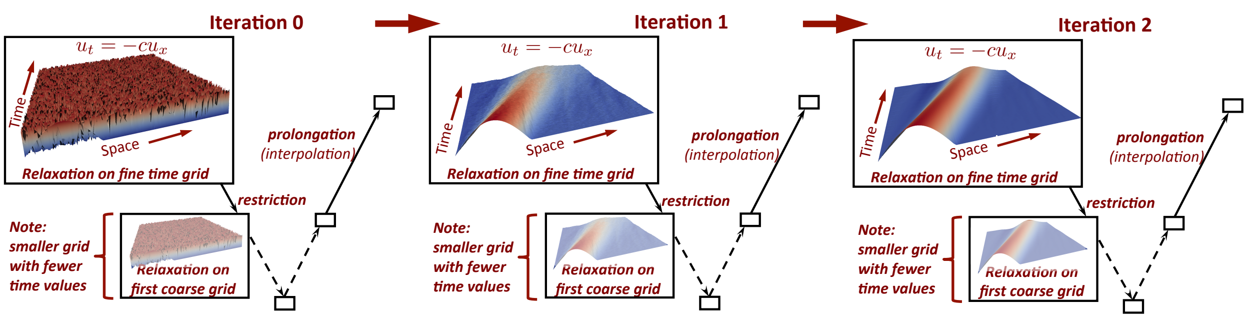


Figure 2: χ Braid iterations.

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,

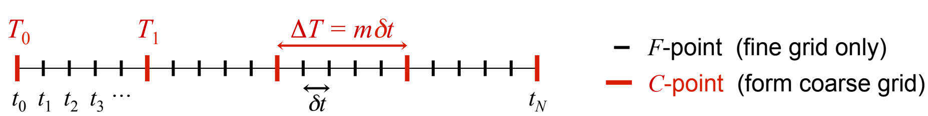
$$A\mathbf{u} \equiv \begin{pmatrix} I & & & \\ -\Phi_1 & I & & \\ & \ddots & \ddots & \\ & & -\Phi_N & I \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_N \end{pmatrix} \equiv \mathbf{g}$$

or

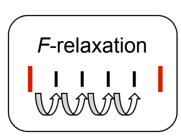
$$A\mathbf{u} = \mathbf{g}.$$

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



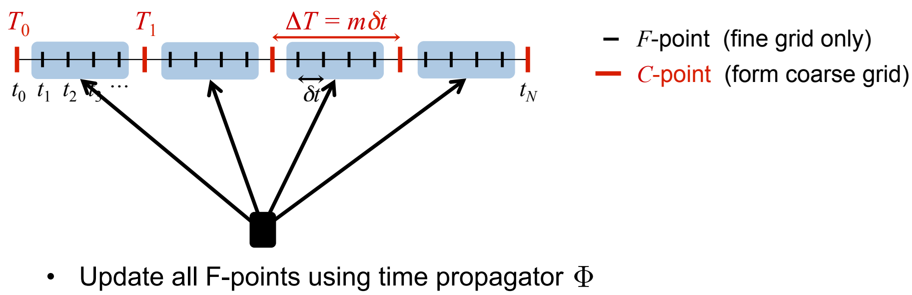
is like line-relaxation in space in that the residual is set to 0 for an entire time step). An F sweep simply updates time values by integrating with Φ over all the F-points from one C-point to the next, as depicted next.

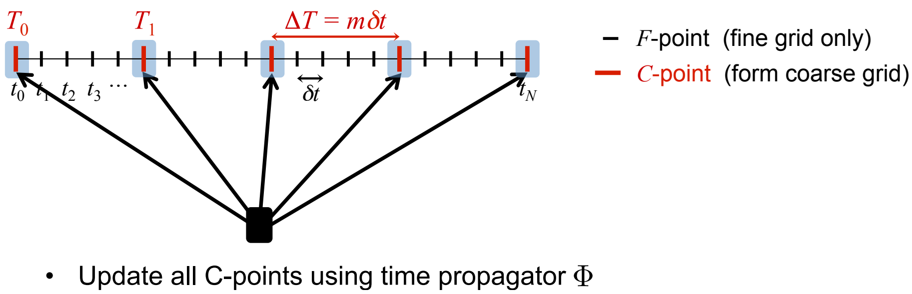


But, such an update can be done simultaneously over all F intervals in parallel, as depicted next.

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.





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.

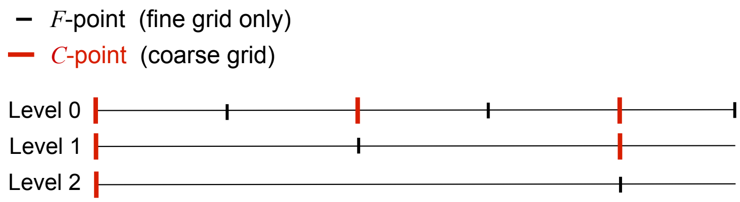
1. Relax on $A(\mathbf{u}) = \mathbf{g}$ using FCF-relaxation
2. Restrict the fine grid approximation and its residual: $\mathbf{u}_\Delta \leftarrow R\mathbf{u}, \quad \mathbf{r}_\Delta \leftarrow R(\mathbf{g} - A(\mathbf{u})$
3. Solve $A_\Delta(\mathbf{v}_\Delta) = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$
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 $m^2 < 6 \le m^3$. 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 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.

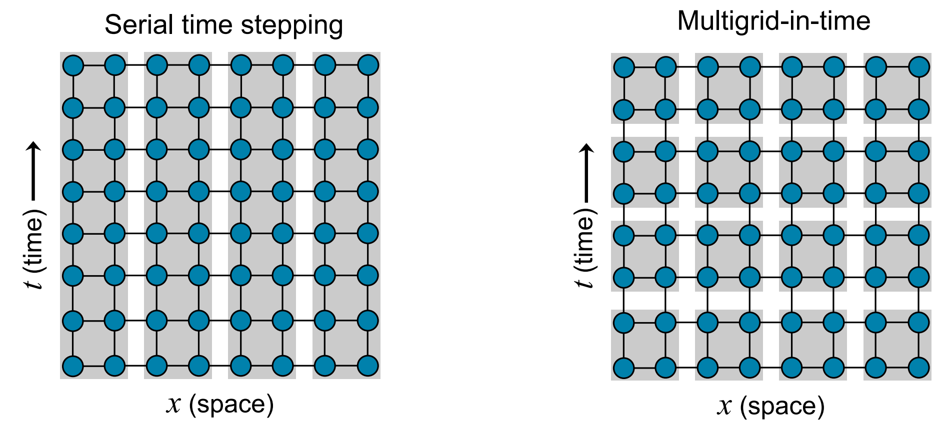
χBraid

- 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_`

2.3.1 Parallel decomposition and memory

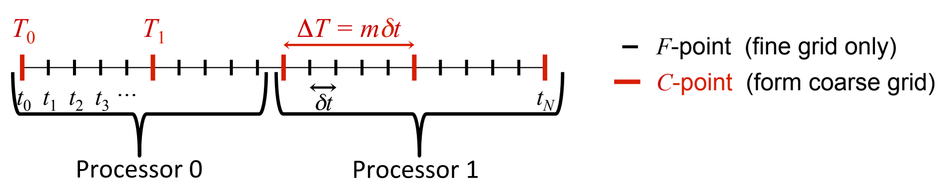
- χBraid decomposes the problem in parallel as depicted next. As you can see, traditional time stepping only

[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



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.

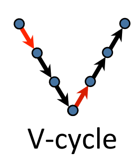
- χBraid only handles temporal parallelism and is agnostic to the spatial decomposition. See braid_Split-Commworld. Each processor owns a certain number of CF intervals of points, as depicted next, where each processor owns 2 CF intervals. χBraid distributes Intervals evenly on the finest grid.



- 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

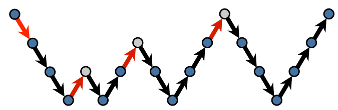
There are two main cycling strategies available in χBraid, F-and V-cycles. These two cycles differ in how often and the order in which coarse levels are visited. A V-cycle is depicted next, and is a simple recursive application of the Two-Grid Algorithm.



An F-cycle visits coarse grids more frequently and in a different order. Essentially, an F-cycle uses a V-cycle as the

Mathematics 332.7 (2001): 661-668.

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.

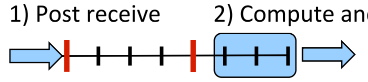
The number of FC relaxation sweeps is another important algorithmic setting. Note that at least one F-relaxation sweep is always done on a level. A few summary points about relaxation are as follows.

- Using FCF (or even FCFCF, FCFCFCF) relaxation, corresponding to passing *braid_SetNRelax* a value of 1, 2 or 3 respectively, will result in a 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 \form#0Braid Hierarchy

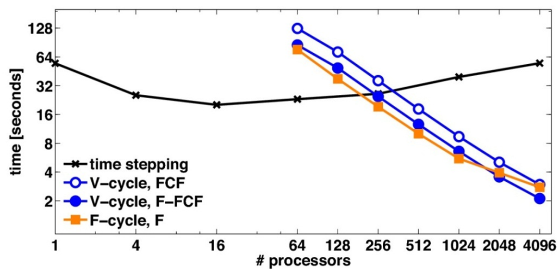
Some of the more basic χBraid function calls allow you to control aspects discussed here.

- 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 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;
}
```

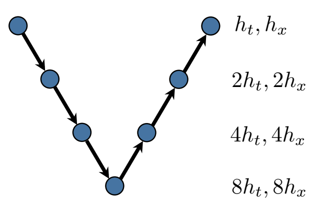
7. **Write**: This function tells χBraid how to write a vector at time *t* to screen, file, etc... The user defines what is appropriate output. Notice how you are told the time value of the vector *u* and even more information in *status*. This lets you tailor the output to only certain time values.
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 Coding Style

Code should follow the `ellemtel` style. See braid/misc/sample_c_code.c, and for emacs and vim style files, see braid/misc/sample.vimrc, and braid/misc/sample.emacs.

7 Using Doxygen

To build the documentation, doxygen must be version 1.8 or greater. χBraid documentation uses a `markdown` syntax both in source file comments and in $*$.md files.

To make the documentation,

```
$ make user_manual
$ acroread user_manual.pdf
```

or to make a more extensive reference manual for developers,

```
$ make developer_manual
$ acroread developer_manual.pdf
```

Developers can run doxygen from a precompiled binary, which may or may not work for your machine,

```
/usr/casc/hypre/braid/share/doxygen/bin/doxygen
```

or build doxygen from

```
/usr/casc/hypre/braid/share/doxygen.tgz
```

- Compiling doxygen requires a number of dependencies like Bison, GraphViz and Flex. Configure will tell you what you're missing
- Unpack doxygen.tgz, then from the doxygen directory

```
./configure --prefix some_dir_in_your_path
make
make install
```

Documentation Strategy

- The doxygen comments are to be placed in the header files.
- A sample function declaration using the documenation approach using markdown (including typesetting equations) is in braid.h for the function braid_Init()
- A sample structure documentation is in _braid.h for _braid_Core_struct
- Descriptors for files can also be added, as at the top of braid.h
- The Doxygen manual is at `http://www.stack.nl/~dimitri/doxygen/manual/index.html`

χBraid Doxygen details

The user and developer manuals are ultimately produced by Latex. The formatting of the manuals is configured according to the following.

- docs/local_doxygen.sty
  - Latex style file used
- docs/user_manual_header.tex
  - User manual title page and header info
- docs/developer_manual_header.tex
  - Developer manual title page and header info
- $*$.md
  - Any file ending in .md is extra documentation in markdown format, like Introduction.md or the various Readme.md files in each directory. This material can be read in plain-text or when it's compiled by Doxygen and Latex.
- docs/user_manual.conf
  - Doxygen configure file for the user manual
  - The FILE_NAMES tag is a filter to only include the user interface routines in braid.h
  - The INPUT tag orders the processing of the files and hence the section ordering
- docs/reference_manual.conf
  - Same as user_manual.conf, but the FILE_NAMES tag does not exclude any file from processing.
- docs/img
  - Contains the images
- To regenerate generic doxygen latex files, type
  `$ doxygen -w latex header.tex footer.tex doxygen.sty doxy.conf`
  If this is done, then the .conf file must be changed to use the new header file and to copy the local_doxygen.sty file to the latex directory.

8 Regression Testing

Overview

- There are three levels in the testing framework. At each level, the fine-grain output from a `testscript.sh` is dumped into a directory `testscript.dir`, with the standard out and error stored in `testscript.out` and `testscript.err`. The test `testscript.sh` passes if `testscript.err` is empty (nothing is written to standard error).

- Basic instructions: run a test with a command like
  ```
  $ ./test.sh diffusion2D.sh
  ```
  Then, see if `diffusion2D.err` is of size 0. If it is not, look at it's contents to see which test failed.
- To add a new regression test, create a new lowest level script like `diffusion2D.sh` and then call it from a machine script at level 2.
- Regression tests should be run before pushing code. It is recommended to run the basic (lowest level) tests like `diffusion2d.sh` or machine test like `machine-tux.sh`

Lowest Level Test Scripts

As an example, here we look at one of the lowest level tests, the diffusion2d test.

Files used:

- `test.sh`
- `diffusion2D.sh`
- `diffusion2D.saved`

Output:

- `diffusion2D.dir`
- `diffusion2D.err`
- `diffusion2D.out`

At this level, we execute

```
$ ./test.sh diffusion2D.sh
```

or just

```
$ ./diffusion2D.sh
```

The script `diffusion2D.sh` must create `diffusion2D.dir` and place all fine-grain test output in this directory. `test.sh` captures the standard out and error in `diffusion2D.out` and `diffusion2D.err`. The test `diffusion2D.sh` passes if `diffusion2D.err` is empty (nothing is written to standard error).

The strategy for low level scripts like `diffusion2D.sh` is to run a sequence of test drivers such as

```
$ mpirun -np 1 ../examples/drive-02 -pgrid 1 1 1 -nt 256
$ mpirun -np 4 ../examples/drive-02 -pgrid 1 1 4 -nt 256
```

The output from the first mpirun test must then be written to files named

```
diffusion2D.dir/unfiltered.std.out.0
diffusion2D.dir/std.out.0
diffusion2D.dir/std.err.0
```

and the second mpirun test similarly writes the files

```
diffusion2D.dir/unfiltered.std.out.1
diffusion2D.dir/std.out.1
diffusion2D.dir/std.err.1
```

Subsequent tests are written to higher numbered files. The `unfiltered.std.out.num` file contains all of the standard out for the test, while `std.out.num` contains filtered output (usually from a `grep` command) and could contain the output lines such as iteration numbers and number of levels. The file `std.err.num` contains the standard error output.

To see if a test ran correctly, `std.out.num` is compared to saved output in `diffusion2D.saved`. The file `diffusion2D.saved` contains the concatenated output from all the tests that `diffusion2D.sh` will run. For the above example, this file could look like

```
# Begin Test 1
number of levels     = 6
iterations           = 16
# Begin Test 2
number of levels     = 4
iterations           = 8
```

This saved output is split into an individual file for each test (using `# Begin Test` as a delimiter) and these new files are placed in `diffusion2D.dir`. So, after running these two regression tests, `diffusion2D.dir` will contain

```
diffusion2D.saved.0
diffusion2D.saved.1
unfiltered.std.out.0
std.out.0
std.err.0
unfiltered.std.out.1
std.out.1
std.err.1
```

An individual test has passed if `std.err.num` is empty. The file `std.err.num` contains a diff between `diffusion2D.save.num` and `std.out.num` (the diff ignores whitespace and the delimiter `# Begin Test`).

Last in the directy where you ran `./test.sh diffusion2d.sh`, the files

```
diffusion2D.err
diffusion2D.out
```

will be created. If all the tests passed then `diffusion2D.err` will be empty. Otherwise, it will contain the filenames of the `std.err.num` files that are non-empty, representing failed tests.

Level 2 Scripts

As an example, here we look at one of the Level 2 tests, the machine-tux test that Jacob runs.

Files used:

- `machine-tux.sh`

Output:

- `machine-tux.dir`
- `machine-tux.err` (only generated if `autotest.sh` is used to run `machine-tux.sh`)
- `machine-tux.out` (only generated if `autotest.sh` is used to run `machine-tux.sh`)

At this level, we execute

```
./machine-tux.sh
```

The autotest framework (`autotest.sh`) calls machine scripts in this way. Each machine script should be short and call lower-level scripts like `diffusion2D.sh`. The output from lower-level scripts must be moved to `machine-tux.dir` like this:

```
$ ./test.sh diffusion2D.sh
$ mv -f diffusion2D.dir machine-tux.dir
$ mv -f diffusion2D.out machine-tux.dir
$ mv -f diffusion2D.err machine-tux.dir
```

All error files from `diffusion2D.sh` will be placed in `machine-tux.dir`, so if `machine-tux.dir` has all zero `*.err` files, then the `machine-tux` test has passed.

To begin testing on a new machine, like vulcan, add a new machine script similar to `machine-tux.sh` and change `autotest.sh` to recognize and run the new machine test. To then use `autotest.sh` with the machine script, you'll have to set up a passwordless connection from the new machine to

```
/usr/casc/hypre/braid/testing
```

Level 3 Script

Here we look at the highest level, where `autotest.sh` runs all of the level 2 machine tests and emails out the results.

Files used:

- `autotest.sh`

Output:

- `test/autotest_finished`
- `/usr/casc/hypre/braid/testing/AUTOTEST-20**.**.**-Day`
- Email to recipients listed in `autotest.sh`

At the highest level sits `autotest.sh` and is called automatically as a cronjob. If you just want to check to see if you've broken anything with a commit, just use lower level scripts.

There are four steps to running autotest.

- Step 1
  ```
  $ ./autotesh.sh -init
  ```
  will do a pull from master for the current working repository and recompile Braid. The autotest output files (`autotest.err` and `autotest.out`) and the output directory (autotest_finished) are initialized.
- Step 2
  ```
  $ ./autotest.sh -tux343
  ```
  will run the autotests on tux343. This command will look for a `machine-tux.sh`, and execute it, moving the resulting
  ```
  machine-tux.dir
  machine-tux.err
  machine-tux.out
  ```
  into `test/autotest_finished`.
- Step 3
  ```
  $ ./autotest.sh -remote-copy
  ```
  will copy `/test/autotest_finished/*` to a time-stamped directory such as
  `/usr/casc/hypre/braid/testing/AUTOTEST-2013.11.18-Mon`
  Alternatively,
  ```
  $ ./autotesh.sh -remote-copy tux343
  ```
  will ssh through tux343 to copy to `/usr/casc`. Multiple machines may independently be running regression tests and then copy to `AUTOTEST-2013.11.18-Mon`.
- Step 4
  ```
  $ ./autotest.sh -summary-email
  ```
  will email everyone listed in the `$email_list` (an `autotest.sh` variable)

Cronfile

To add entries to your crontab, First, put your new cronjob lines into `cronfile`. Then see what you already have in your crontab file with

```
$ crontab -l
```

Next, append to cronfile whatever you already have

```
$ crontab -l >> cronfile
```

Finally, tell crontab to use your cronfile

```
$ crontab cronfile
```

Then make sure it took affect with

```
$ crontab -l
```

Crontab entry format uses '*' to mean "every" and '*/m' to mean "every m-th". The first five entries on each line correspond respectively to:

- minute (0-56)
- hour (0-23)
- day of month (1-31)
- month (1-12)
- day of week (0-6)(0=Sunday)

Jacob's crontab (on tux343):

```
05 14 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -init
05 14 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -tux343
05 14 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -remote-c
05 14 * * * source /etc/profile; source $HOME/.bashrc; cd $HOME/joint_repos/braid/test; ./autotest.sh -summary-
```

9 Module Index

9.1 Modules

Here is a list of all modules:

User-written routines 23

User interface routines 26

Braid test routines 35

10 Data Structure Index

10.1 Data Structures

Here are the data structures with brief descriptions:

_braid_AccuracyHandle 39

_braid_CommHandle 39

_braid_Core 40

_braid_Grid 44

_braid_Status 46

11 File Index

11.1 File List

Here is a list of all files with brief descriptions:

_braid.h
Define headers for developer routines 47

braid.h
Define headers for user interface routines 53

braid_test.h
Define headers for Braid test routines 55

util.h
Define headers for utility routines 55

12 Module Documentation

12.1 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)

12.1.1 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.

12.1.2 Typedef Documentation

12.1.2.1 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.

12.1.2.2 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.

12.1.2.3 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.

12.1.2.4 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.

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

Clone *u* into *v_ptr*

12.1.2.6 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.

12.1.2.7 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* = $<u, v>$

12.1.2.8 typedef **braid_Int**(∗ braid_PtFcnFree)(**braid_App** app,**braid_Vector** u)

Free and deallocate *u*

12.1.2.9 typedef braid_Int(∗ braid_PtFcnInit)(braid_App app,braid_Real t,braid_Vector ∗u_ptr)

Initializes a vector *u_ptr* at time *t*

12.1.2.10 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.

12.1.2.11 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.

12.1.2.12 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*

12.1.2.13 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.

12.1.2.14 typedef struct _braid_Status_struct∗ braid_Status

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

12.1.2.15 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.

12.2 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)

12.2.1 Detailed Description

these are interface routines to initialize and run Braid

12.2.2 Typedef Documentation

12.2.2.1 typedef struct _braid_Core_struct∗ **braid_Core**

points to the core structure defined in _braid.h

12.2.3 Function Documentation

12.2.3.1 braid_Int braid_Destroy (braid_Core *core*)

Clean up and destroy core.

Parameters

core	braid_Core (_braid_Core) struct

12.2.3.2 braid_Int braid_Drive (braid_Core *core*)

Carry out a simulation with Braid. Integrate in time.

Parameters

core	braid_Core (_braid_Core) struct

12.2.3.3 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

12.2.3.4 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

12.2.3.5 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

12.2.3.6 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

12.2.3.7 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

12.2.3.8 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

12.2.3.9 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

12.2.3.10 braid_Int braid_PrintStats (braid_Core *core*)

Print statistics after a Braid run.

Parameters

core	braid_Core (_braid_Core) struct

12.2.3.11 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

12.2.3.12 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

12.2.3.13 braid_Int braid_SetFMG (braid_Core *core*)

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

Parameters

core	braid_Core (_braid_Core) struct

12.2.3.14 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

12.2.3.15 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

12.2.3.16 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

12.2.3.17 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

12.2.3.18 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

12.2.3.19 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*

12.2.3.20 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

12.2.3.21 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

12.2.3.22 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

12.2.3.23 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

12.2.3.24 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

12.2.3.25 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

12.2.3.26 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

12.2.3.27 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)

12.3 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)

12.3.1 Detailed Description

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

12.3.2 Function Documentation

12.3.2.1 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.

12.3.2.2 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

12.3.2.3 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

12.3.2.4 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

12.3.2.5 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

12.3.2.6 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

12.3.2.7 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

13 Data Structure Documentation

13.1 _braid_AccuracyHandle Struct Reference

Data Fields

- braid_Int matchF
- braid_Real value
- braid_Real old_value
- braid_Real loose
- braid_Real tight
- braid_Int tight_used

13.1.1 Detailed Description

Braid Accuracy Handle, used for controlling the accuracy of solves during implicit time stepping. For instance, to do less accurate solves on coarse time grids

13.1.2 Field Documentation

13.1.2.1 braid_Real loose

loose accuracy for spatial solves

13.1.2.2 braid_Int matchF

13.1.2.3 braid_Real old_value

old accuracy value used in FRestrict

13.1.2.4 braid_Real tight

tight accuracy for spatial solves

13.1.2.5 braid_Int tight_used

tight accuracy used (1) or not (0)

13.1.2.6 braid_Real value

accuracy value

The documentation for this struct was generated from the following file:

- _braid.h

13.2 _braid_CommHandle Struct Reference

Data Fields

- braid_Int request_type
- braid_Int num_requests

- MPI_Request * requests
- MPI_Status * status
- void * buffer
- braid_Vector * vector_ptr

13.2.1 Detailed Description

Braid comm handle structure

Used for initiating and completing nonblocking communication to pass braid_Vectors between processors.

13.2.2 Field Documentation

13.2.2.1 void* buffer

Buffer for message

13.2.2.2 braid_Int num_requests

number of active requests for this handle, usually 1

13.2.2.3 braid_Int request_type

two values: recv type = 1, and send type = 0

13.2.2.4 MPI_Request* requests

MPI request structure

13.2.2.5 MPI_Status* status

MPI status

13.2.2.6 braid_Vector* vector_ptr

braid_vector being sent/received

The documentation for this struct was generated from the following file:

- _braid.h

13.3 _braid_Core Struct Reference

Data Fields

- 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_PtFcnCoarsen coarsen
- braid_PtFcnRefine refine
- braid_Int write_level
- braid_Int print_level
- braid_Int max_levels
- braid_Int max_coarse
- braid_Real tol
- braid_Int rtol
- braid_Int ∗ nrels
- braid_Int nrdefault
- braid_Int ∗ cfactors
- braid_Int cfdefault
- braid_Int max_iter
- braid_Int niter
- braid_Real rnorm
- braid_Int fmg
- braid_Int nfmg_Vcyc
- _braid_AccuracyHandle ∗ accuracy
- braid_Int gupper
- braid_Int ∗ rfactors
- braid_Int nlevels
- _braid_Grid ∗∗ grids
- braid_Real localtime
- braid_Real globaltime

13.3.1 Detailed Description

The typedef _braid_Core struct is a **critical** part of Braid and is passed to *each* routine in Braid. It thus allows each routine access to Braid attributes.

13.3.2 Field Documentation

13.3.2.1 _braid_AccuracyHandle∗ accuracy

accuracy of spatial solves on different levels

13.3.2.2 braid_App app

application data for the user

13.3.2.3 braid_PtFcnBufPack bufpack

pack a buffer

13.3.2.4 braid_PtFcnBufSize bufsize

return buffer size

13.3.2.5 braid_PtFcnBufUnpack bufunpack

unpack a buffer

13.3.2.6 braid_Int* cfactors

coarsening factors

13.3.2.7 braid_Int cfdefault

default coarsening factor

13.3.2.8 braid_PtFcnClone clone

clone a vector

13.3.2.9 braid_PtFcnCoarsen coarsen

(optional) return a coarsened vector

13.3.2.10 MPI_Comm comm

communicator for the time dimension

13.3.2.11 MPI_Comm comm_world

13.3.2.12 braid_PtFcnDot dot

dot product

13.3.2.13 braid_Int fmg

use FMG cycle

13.3.2.14 braid_PtFcnFree free

free up a vector

13.3.2.15 braid_Real globaltime

global wall time for braid_Drive()

13.3.2.16 _braid_Grid** grids

pointer to temporal grid structures for each level

13.3.2.17 braid_Int gupper

global upper index on the fine grid

13.3.2.18 braid_PtFcnInit init

return an initialized braid_Vector

13.3.2.19 braid_Real localtime

local wall time for braid_Drive()

13.3.2.20 braid_Int max_coarse

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

13.3.2.21 braid_Int max_iter

maximum number of multigrid in time iterations

13.3.2.22 braid_Int max_levels

maximum number of temporal grid levels

13.3.2.23 braid_Int nfmg_Vcyc

number of V-cycle calls at each level in FMG

13.3.2.24 braid_Int niter

number of iterations

13.3.2.25 braid_Int nlevels

number of temporal grid levels

13.3.2.26 braid_Int nrdefault

default number of pre-relaxations

13.3.2.27 braid_Int* nrels

number of pre-relaxations on each level

13.3.2.28 braid_Int ntime

initial number of time intervals

13.3.2.29 braid_PtFcnPhi phi

apply phi function

13.3.2.30 braid_Int print_level

determines amount of output printed to screem (0,1,2)

13.3.2.31 braid_PtFcnRefine refine

(optional) return a refined vector

13.3.2.32 braid_Int* rfactors

refinement factors for finest grid (if any)

13.3.2.33 braid_Real rnorm

residual norm

13.3.2.34 braid_Int rtol

use relative tolerance

13.3.2.35 braid_PtFcnSum sum

vector sum

13.3.2.36 braid_Real tol

stopping tolerance

13.3.2.37 braid_Real tstart

start time

13.3.2.38 braid_Real tstop

stop time

13.3.2.39 braid_PtFcnWrite write

write the vector

13.3.2.40 braid_Int write_level

determines how often to call the user's write routine

The documentation for this struct was generated from the following file:

- _braid.h

13.4 _braid_Grid Struct Reference

Data Fields

- braid_Int level
- braid_Int ilower
- braid_Int iupper
- braid_Int clower
- braid_Int cupper
- braid_Int cfactor
- braid_Int ncpoints
- braid_Vector * ua
- braid_Real * ta
- braid_Vector * va
- braid_Vector * wa
- braid_Int recv_index
- braid_Int send_index
- _braid_CommHandle * recv_handle
- _braid_CommHandle * send_handle

- braid_Vector * ua_alloc
- braid_Real * ta_alloc
- braid_Vector * va_alloc
- braid_Vector * wa_alloc

13.4.1 Detailed Description

Braid Grid structure for a certain time level

Holds all the information for a processor related to the temporal grid at this level.

13.4.2 Field Documentation

13.4.2.1 braid_Int cfactor

coarsening factor

13.4.2.2 braid_Int clower

smallest C point index

13.4.2.3 braid_Int cupper

largest C point index

13.4.2.4 braid_Int ilower

smallest time index at this level

13.4.2.5 braid_Int iupper

largest time index at this level

13.4.2.6 braid_Int level

Level that grid is on

13.4.2.7 braid_Int ncpoints

number of C points

13.4.2.8 _braid_CommHandle* recv_handle

Handle for nonblocking receives of braid_Vectors

13.4.2.9 braid_Int recv_index

-1 means no receive

13.4.2.10 _braid_CommHandle* send_handle

Handle for nonblocking sends of braid_Vectors

13.4.2.11 braid_Int send_index

-1 means no send

13.4.2.12 braid_Real∗ ta

time values (all points)

13.4.2.13 braid_Real∗ ta_alloc

original memory allocation for ta

13.4.2.14 braid_Vector∗ ua

unknown vectors (C-points only)

13.4.2.15 braid_Vector∗ ua_alloc

original memory allocation for ua

13.4.2.16 braid_Vector∗ va

restricted unknown vectors (all points, NULL on level 0)

13.4.2.17 braid_Vector∗ va_alloc

original memory allocation for va

13.4.2.18 braid_Vector∗ wa

rhs vectors f-v (all points, NULL on level 0)

13.4.2.19 braid_Vector∗ wa_alloc

original memory allocation for wa

The documentation for this struct was generated from the following file:

- _braid.h

13.5 _braid_Status Struct Reference

Data Fields

- braid_Int iter
- braid_Int level
- braid_Real rnorm
- braid_Int done

13.5.1 Detailed Description

Points to the status structure which defines the status of Braid at a given instant on a some level during a run. The user accesses it through braid_Get∗∗Status() functions.

13.5.2 Field Documentation

13.5.2.1 braid_Int done

boolean describing whether Braid has finished

13.5.2.2 braid_Int iter

Braid iteration number

13.5.2.3 braid_Int level

current level in Braid

13.5.2.4 braid_Real rnorm

residual norm

The documentation for this struct was generated from the following file:

- _braid.h

14 File Documentation

14.1 _braid.h File Reference

Data Structures

- struct _braid_Status
- struct _braid_CommHandle
- struct _braid_AccuracyHandle
- struct _braid_Grid
- struct _braid_Core

Macros

- #define _braid_CommHandleElt(handle, elt) ((handle) -> elt)
- #define _braid_GridElt(grid, elt) ((grid) -> elt)
- #define _braid_StatusElt(status, elt) ((status) -> elt)
- #define _braid_CoreElt(core, elt) ((core) -> elt)
- #define _braid_CoreFcn(core, fcn) (*((core) -> fcn))
- #define _braid_TAlloc(type, count) ((type *)malloc((size_t)(sizeof(type) * (count))))
- #define _braid_CTAlloc(type, count) ((type *)calloc((size_t)(count), (size_t)sizeof(type)))
- #define _braid_TReAlloc(ptr, type, count) ((type *)realloc((char *)ptr, (size_t)(sizeof(type) * (count))))
- #define _braid_TFree(ptr) (free((char *)ptr), ptr = NULL)
- #define _braid_MapFineToCoarse(findex, cfactor, cindex) (cindex = (findex)/(cfactor))
- #define _braid_MapCoarseToFine(cindex, cfactor, findex) (findex = (cindex)*(cfactor))
- #define _braid_IsFPoint(index, cfactor) ((index)%(cfactor))
- #define _braid_IsCPoint(index, cfactor) (!_braid_IsFPoint(index, cfactor))

Functions

- braid_Int _braid_GetDistribution (braid_Core core, braid_Int *ilower_ptr, braid_Int *iupper_ptr)
- braid_Int _braid_GetProc (braid_Core core, braid_Int level, braid_Int index, braid_Int *proc_ptr)
- braid_Int _braid_CommRecvInit (braid_Core core, braid_Int level, braid_Int index, braid_Vector *vector_ptr, _braid_CommHandle **handle_ptr)

- braid_Int _braid_CommSendInit (braid_Core core, braid_Int level, braid_Int index, braid_Vector vector, _braid_CommHandle **handle_ptr)
- braid_Int _braid_CommWait (braid_Core core, _braid_CommHandle **handle_ptr)
- braid_Int _braid_UCommInit (braid_Core core, braid_Int level)
- braid_Int _braid_UCommInitF (braid_Core core, braid_Int level)
- braid_Int _braid_UCommWait (braid_Core core, braid_Int level)
- braid_Int _braid_UGetInterval (braid_Core core, braid_Int level, braid_Int interval_index, braid_Int *flo_ptr, braid_Int *fhi_ptr, braid_Int *ci_ptr)
- braid_Int _braid_UGetVectorRef (braid_Core core, braid_Int level, braid_Int index, braid_Vector *u_ptr)
- braid_Int _braid_USetVectorRef (braid_Core core, braid_Int level, braid_Int index, braid_Vector u)
- braid_Int _braid_UGetVector (braid_Core core, braid_Int level, braid_Int index, braid_Vector *u_ptr)
- braid_Int _braid_USetVector (braid_Core core, braid_Int level, braid_Int index, braid_Vector u)
- braid_Int _braid_UWriteVector (braid_Core core, braid_Int level, braid_Int index, braid_Status status, braid_Vector u)
- braid_Int _braid_Phi (braid_Core core, braid_Int level, braid_Int index, braid_Real accuracy, braid_Vector u, braid_Int *rfactor)
- braid_Int _braid_Step (braid_Core core, braid_Int level, braid_Int index, braid_Real accuracy, braid_Vector u)
- braid_Int _braid_Coarsen (braid_Core core, braid_Int level, braid_Int f_index, braid_Int c_index, braid_Vector fvector, braid_Vector *cvector)
- braid_Int _braid_Refine (braid_Core core, braid_Int level, braid_Int f_index, braid_Int c_index, braid_Vector cvector, braid_Vector *fvector)
- braid_Int _braid_GridInit (braid_Core core, braid_Int level, braid_Int ilower, braid_Int iupper, _braid_Grid **grid_ptr)
- braid_Int _braid_GridDestroy (braid_Core core, _braid_Grid *grid)
- braid_Int _braid_InitGuess (braid_Core core, braid_Int level)
- braid_Int _braid_CFRelax (braid_Core core, braid_Int level)
- braid_Int _braid_FRestrict (braid_Core core, braid_Int level, braid_Int iter, braid_Real *rnorm_ptr)
- braid_Int _braid_FInterp (braid_Core core, braid_Int level, braid_Int iter, braid_Real rnorm)
- braid_Int _braid_FRefine (braid_Core core, braid_Int *refined_ptr)
- braid_Int _braid_FWrite (braid_Core core, braid_Real rnorm, braid_Int iter, braid_Int level, braid_Int done)
- braid_Int _braid_InitHierarchy (braid_Core core, _braid_Grid *fine_grid)
- braid_Int _braid_InitStatus (braid_Real rnorm, braid_Int iter, braid_Int level, braid_Int done, braid_Status *status_ptr)
- braid_Int _braid_DestroyStatus (braid_Status status)

Variables

- braid_Int _braid_error_flag
- FILE * _braid_printfile

14.1.1 Detailed Description

Define headers for developer routines. This file contains the headers for developer routines.

14.1.2 Macro Definition Documentation

14.1.2.1 #define _braid_CommHandleElt(*handle,* *elt*) ((handle) -> elt)

Accessor for _braid_CommHandle attributes

14.1.2.2 #define _braid_CoreElt(core, elt) ((core) -> elt)

Accessor for _braid_Core attributes

14.1.2.3 #define _braid_CoreFcn(core, fcn) (*((core) -> fcn))

Accessor for _braid_Core functions

14.1.2.4 #define _braid_CTAlloc(type, count) ((type *)calloc((size_t)(count), (size_t)sizeof(type)))

Allocation macro

14.1.2.5 #define _braid_GridElt(grid, elt) ((grid) -> elt)

Accessor for _braid_Grid attributes

14.1.2.6 #define _braid_IsCPoint(index, cfactor) (!_braid_IsFPoint(index, cfactor))

Boolean, returns whether a time index is an C-point

14.1.2.7 #define _braid_IsFPoint(index, cfactor) ((index)%(cfactor))

Boolean, returns whether a time index is an F-point

14.1.2.8 #define _braid_MapCoarseToFine(cindex, cfactor, findex) (findex = (cindex)*(cfactor))

Map a coarse time index to a fine time index, assumes a uniform coarsening factor.

14.1.2.9 #define _braid_MapFineToCoarse(findex, cfactor, cindex) (cindex = (findex)/(cfactor))

Map a fine time index to a coarse time index, assumes a uniform coarsening factor.

14.1.2.10 #define _braid_StatusElt(status, elt) ((status) -> elt)

Accessor for _braid_Status attributes

14.1.2.11 #define _braid_TAlloc(type, count) ((type *)malloc((size_t)(sizeof(type) * (count))))

Allocation macro

14.1.2.12 #define _braid_TFree(ptr) (free((char *)ptr), ptr = NULL)

Free memory macro

14.1.2.13 #define _braid_TReAlloc(ptr, type, count) ((type *)realloc((char *)ptr, (size_t)(sizeof(type) * (count))))

Re-allocation macro

14.1.3 Function Documentation

14.1.3.1 braid_Int _braid_CFRelax (braid_Core core, braid_Int level)

Do nu sweeps of F-then-C relaxation on *level*

14.1.3.2 braid_Int _braid_Coarsen (braid_Core *core,* braid_Int *level,* braid_Int *f_index,* braid_Int *c_index,* braid_Vector *fvector,* braid_Vector ∗ *cvector*)

Coarsen in space on *level* by calling the user's coarsen function. The vector corresponding to the time step index *f_index* on the fine grid is coarsened to the time step index *c_index* on the coarse grid. The output goes in *cvector* and the input vector is *fvector*.

14.1.3.3 braid_Int _braid_CommRecvInit (braid_Core *core,* braid_Int *level,* braid_Int *index,* braid_Vector ∗ *vector_ptr,* _braid_CommHandle ∗∗ *handle_ptr*)

Initialize a receive to go into *vector_ptr* for the given time *index* on *level*. Also return a comm handle *handle_ptr* for querying later, to see if the receive has occurred.

14.1.3.4 braid_Int _braid_CommSendInit (braid_Core *core,* braid_Int *level,* braid_Int *index,* braid_Vector *vector,* _braid_CommHandle ∗∗ *handle_ptr*)

Initialize a send of *vector* for the given time *index* on *level*. Also return a comm handle *handle_ptr* for querying later, to see if the send has occurred.

14.1.3.5 braid_Int _braid_CommWait (braid_Core *core,* _braid_CommHandle ∗∗ *handle_ptr*)

Block on the comm handle *handle_ptr* until the MPI operation (send or recv) has completed

14.1.3.6 braid_Int _braid_DestroyStatus (braid_Status *status*)

Destroy a braid_Status structure

14.1.3.7 braid_Int _braid_FInterp (braid_Core *core,* braid_Int *level,* braid_Int *iter,* braid_Real *rnorm*)

F-Relax on *level* and interpolate to *level-1*

The output is set in the braid_Grid in core, so that the vector *u* on *level* is created by interpolating from *level+1*.

If the user has set spatial refinement, then this user-defined routine is also called.

Parameters

core	braid_Core (_braid_Core) struct
level	interp from level to level+1
iter	current iteration number (for user info)
rnorm	residual norm (if level 0)

14.1.3.8 braid_Int _braid_FRefine (braid_Core *core,* braid_Int ∗ *refined_ptr*)

Create a new fine grid based on user refinement factor information, then F-relax and interpolate to the new fine grid and create a new multigrid hierarchy. In general, this will require load re-balancing as well.

RDF: Todo, routine is unwritten

14.1.3.9 braid_Int _braid_FRestrict (braid_Core *core,* braid_Int *level,* braid_Int *iter,* braid_Real ∗ *rnorm_ptr*)

F-Relax on *level* and then restrict to *level+1*

The output is set in the braid_Grid in core, so that the restricted vectors *va* and *wa* will be created, representing *level+1* versions of the unknown and rhs vectors.

If the user has set spatial coarsening, then this user-defined routine is also called.

If *level==0*, then *rnorm_ptr* will contain the residual norm.

Parameters

core	braid_Core (_braid_Core) struct
level	restrict from level to level+1
iter	current iteration number (for user info)
rnorm_ptr	pointer to residual norm (if level 0)

14.1.3.10 braid_Int _braid_FWrite (braid_Core *core,* braid_Real *rnorm,* braid_Int *iter,* braid_Int *level,* braid_Int *done*)

Write out the solution on grid *level* at Braid iteration *iter*.

rnorm denotes the last computed residual norm, and *done* is a boolean indicating whether Braid has finished iterating and this is the last Write call.

14.1.3.11 braid_Int _braid_GetDistribution (braid_Core *core,* braid_Int ∗ *ilower_ptr,* braid_Int ∗ *iupper_ptr*)

Determine processor distribution. This must agree with GetProc(). For the processor rank calling this function, it returns the smallest and largest time indices (*ilower_ptr* and *iupper_ptr*) that belong to that processor (the indices may ∗ be F or C points).

14.1.3.12 braid_Int _braid_GetProc (braid_Core *core,* braid_Int *level,* braid_Int *index,* braid_Int ∗ *proc_ptr*)

Return the processor number in *proc_ptr* on which the time step *index* lives for the given *level*. ∗ Returns -1 if *index* is out of range

14.1.3.13 braid_Int _braid_GridDestroy (braid_Core *core,* _braid_Grid ∗ *grid*)

Destroy a Braid *grid*

14.1.3.14 braid_Int _braid_GridInit (braid_Core *core,* braid_Int *level,* braid_Int *ilower,* braid_Int *iupper,* _braid_Grid ∗∗ *grid_ptr*)

Create a new grid object *grid_ptr* in core at *level*

ilower and *iupper* correspond to the lower and upper time index values for this processor on this grid.

14.1.3.15 braid_Int _braid_InitGuess (braid_Core *core,* braid_Int *level*)

Set initial guess at C-points on *level*

14.1.3.16 braid_Int _braid_InitHierarchy (braid_Core *core,* _braid_Grid ∗ *fine_grid*)

Initialize (and re-initialize) hierarchy

14.1.3.17 braid_Int _braid_InitStatus (braid_Real *rnorm,* braid_Int *iter,* braid_Int *level,* braid_Int *done,* braid_Status ∗ *status_ptr*)

Initialize a braid_Status structure in *status_ptr*, setting the status values *rnorm*, *iter*, *level*, *done*

14.1.3.18 braid_Int _braid_Phi (braid_Core *core,* braid_Int *level,* braid_Int *index,* braid_Real *accuracy,* braid_Vector *u,* braid_Int ∗ *rfactor*)

Apply Phi to the vector *u*

This is the vector corresponding to the time step *index* on *level*. *accuracy* is a user set variable to allow for tuning the accuracy of spatial solvesfor implicit stepping. And, *rfactor* allows the user to subdivide time intervals for accuracy purposes.

14.1.3.19 braid_Int _braid_Refine (braid_Core *core,* braid_Int *level,* braid_Int *f_index,* braid_Int *c_index,* braid_Vector *cvector,* braid_Vector ∗ *fvector*)

Refine in space on *level* by calling the user's refine function. The vector corresponding to the time step index *c_index* on the coarse grid is refined to the time step index *f_index* on the fine grid. The output goes in *fvector* and the input vector is *cvector*.

14.1.3.20 braid_Int _braid_Step (braid_Core *core,* braid_Int *level,* braid_Int *index,* braid_Real *accuracy,* braid_Vector *u*)

Integrate one time step at time step *index* to time step *index*+1

14.1.3.21 braid_Int _braid_UCommInit (braid_Core *core,* braid_Int *level*)

Working on all intervals

At *level*, post a receive for the point to the left of ilower (regardless whether ilower is F or C). Then, post a send of iupper if iupper is a C point.

14.1.3.22 braid_Int _braid_UCommInitF (braid_Core *core,* braid_Int *level*)

Working only on F-pt intervals

At *level*, **only** post a receive for the point to the left of ilower if ilower is an F point. Then, post a send of iupper if iupper is a C point.

14.1.3.23 braid_Int _braid_UCommWait (braid_Core *core,* braid_Int *level*)

Finish up communication

On *level*, wait on both the recv and send handles at this level.

14.1.3.24 braid_Int _braid_UGetInterval (braid_Core *core,* braid_Int *level,* braid_Int *interval_index,* braid_Int ∗ *flo_ptr,* braid_Int ∗ *fhi_ptr,* braid_Int ∗ *ci_ptr*)

Retrieve the time step indices at this *level* which correspond to the FC interval given by *interval_index*. *ci_ptr* is the time step index for the C point and *flo_ptr* and *fhi_ptr* are the smallest and largest F point indices in this interval. *flo* = *ci* +1, and *fhi* = *ci* + coarsening_factor - 1

14.1.3.25 braid_Int _braid_UGetVector (braid_Core *core,* braid_Int *level,* braid_Int *index,* braid_Vector ∗ *u_ptr*)

Returns the u-vector in *u_ptr* on grid *level* at point *index*. If *index* is my "receive index" (as set by UCommInit(), for example), the u-vector will be received from a neighbor processor. If *index* is within my index range and is also a C-point, the saved value of u will be used. A NULL value is returned otherwise.

14.1.3.26 braid_Int _braid_UGetVectorRef (braid_Core *core,* braid_Int *level,* braid_Int *index,* braid_Vector ∗ *u_ptr*)

Returns a reference to the local u-vector in *u_ptr* for the grid *level* at point *index*. Caveat: if *index* is not a C-point and within my index range, NULL is returned.

14.1.3.27 braid_Int _braid_USetVector (braid_Core *core,* braid_Int *level,* braid_Int *index,* braid_Vector *u*)

Sets the u-vector on grid *level* at point *index*. If *index* is my "send index" (as set by UCommInit(), for example), a send is initiated to a neighbor processor. If *index* is within my index range and is also a C-point, the value is saved locally.

14.1.3.28 braid_Int _braid_USetVectorRef (braid_Core *core*, braid_Int *level*, braid_Int *index*, braid_Vector *u*)

Stores a reference to the vector *u* on grid *level* at point *index*. If *index* is not a C-point and within this processor's range of time points, then nothing is done.

14.1.3.29 braid_Int _braid_UWriteVector (braid_Core *core*, braid_Int *level*, braid_Int *index*, braid_Status *status*, braid_Vector *u*)

Call the user's write function to write *u* which is the vector corresponding to time step *index* on *level*. *status* holds state information about the current Braid iteration, time value, etc...

14.1.4 Variable Documentation

14.1.4.1 braid_Int _braid_error_flag

This is the global Braid error flag. If it is ever nonzero, an error has occurred.

14.1.4.2 FILE∗ _braid_printfile

This is the print file for redirecting stdout for all Braid screen output

14.2 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)

14.2.1 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.

14.2.2 Typedef Documentation

14.2.2.1 typedef int **braid_Int**

Defines integer type

14.2.2.2 typedef double **braid_Real**

Defines floating point type

14.3 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)

14.3.1 Detailed Description

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

14.4 util.h File Reference

Functions

- braid_Int _braid_ProjectInterval (braid_Int ilower, braid_Int iupper, braid_Int index, braid_Int stride, braid_Int ∗pilower, braid_Int ∗piupper)
- braid_Int _braid_SetAccuracy (braid_Real rnorm, braid_Real loose_tol, braid_Real tight_tol, braid_Real oldAccuracy, braid_Real tol, braid_Real ∗paccuracy)
- braid_Int _braid_printf (const char ∗format,...)
- braid_Int _braid_ParFprintfFlush (FILE ∗file, braid_Int myid, char ∗message,...)

14.4.1 Detailed Description

Define headers for utility routines. This file contains the headers for utility routines. Essentially, if a routine does not take braid_Core (or other Braid specific structs) as an argument, then it's a utility routine.

14.4.2 Function Documentation

14.4.2.1 braid_Int _braid_ParFprintfFlush (FILE ∗ *file,* braid_Int *myid,* char ∗ *message,* *...*)

This is a function that allows for "sane" printing of information in parallel. Currently, only myid = 0 prints, but this can be updated as needs change.

The string *message* is printed and can be multiple parameters in the standard $*$ C-format, like

```
message = '%1.2e is a format string', 1.24
```

14.4.2.2 braid_Int _braid_printf (const char $*$ *format,* *...*)

If set, print to _braid_printfile and then flush. Else print to standard out.

The string *format* can be multiple parameters in the standard $*$ C-format, like

```
format = '%1.2e is a format string', 1.24
```

14.4.2.3 braid_Int _braid_ProjectInterval (braid_Int *ilower,* braid_Int *iupper,* braid_Int *index,* braid_Int *stride,* braid_Int $*$ *pilower,* braid_Int $*$ *piupper*)

Project an interval onto a strided index space that contains the index 'index' and has stride 'stride'. An empty projection is represented by ilower $>$ iupper.

14.4.2.4 braid_Int _braid_SetAccuracy (braid_Real *rnorm,* braid_Real *loose_tol,* braid_Real *tight_tol,* braid_Real *oldAccuracy,* braid_Real *tol,* braid_Real $*$ *paccuracy*)

Determine the accuracy used for the spatial solves based on the ratio of the current residual norm and the stopping tolerance.

Index

_braid.h, 47
  _braid_CFRelax, 49
  _braid_CTAlloc, 49
  _braid_Coarsen, 49
  _braid_CommHandleElt, 48
  _braid_CommRecvInit, 50
  _braid_CommSendInit, 50
  _braid_CommWait, 50
  _braid_CoreElt, 48
  _braid_CoreFcn, 49
  _braid_DestroyStatus, 50
  _braid_FInterp, 50
  _braid_FRefine, 50
  _braid_FRestrict, 50
  _braid_FWrite, 51
  _braid_GetDistribution, 51
  _braid_GetProc, 51
  _braid_GridDestroy, 51
  _braid_GridElt, 49
  _braid_GridInit, 51
  _braid_InitGuess, 51
  _braid_InitHierarchy, 51
  _braid_InitStatus, 51
  _braid_IsCPoint, 49
  _braid_IsFPoint, 49
  _braid_MapCoarseToFine, 49
  _braid_MapFineToCoarse, 49
  _braid_Phi, 51
  _braid_Refine, 51
  _braid_StatusElt, 49
  _braid_Step, 52
  _braid_TAlloc, 49
  _braid_TFree, 49
  _braid_TReAlloc, 49
  _braid_UCommInit, 52
  _braid_UCommInitF, 52
  _braid_UCommWait, 52
  _braid_UGetInterval, 52
  _braid_UGetVector, 52
  _braid_UGetVectorRef, 52
  _braid_USetVector, 52
  _braid_USetVectorRef, 52
  _braid_UWriteVector, 53
  _braid_error_flag, 53
  _braid_printfile, 53
_braid_AccuracyHandle, 39
  loose, 39
  matchF, 39
  old_value, 39
  tight, 39
  tight_used, 39
  value, 39
_braid_CFRelax
  _braid.h, 49
_braid_CTAlloc
  _braid.h, 49
_braid_Coarsen
  _braid.h, 49
_braid_CommHandle, 39
  buffer, 40
  num_requests, 40
  request_type, 40
  requests, 40
  status, 40
  vector_ptr, 40
_braid_CommHandleElt
  _braid.h, 48
_braid_CommRecvInit
  _braid.h, 50
_braid_CommSendInit
  _braid.h, 50
_braid_CommWait
  _braid.h, 50
_braid_Core, 40
  accuracy, 41
  app, 41
  bufpack, 41
  bufsize, 41
  bufunpack, 42
  cfactors, 42
  cfdefault, 42
  clone, 42
  coarsen, 42
  comm, 42
  comm_world, 42
  dot, 42
  fmg, 42
  free, 42
  globaltime, 42
  grids, 42
  gupper, 42
  init, 42
  localtime, 42
  max_coarse, 43
  max_iter, 43
  max_levels, 43
  nfmg_Vcyc, 43
  niter, 43
  nlevels, 43
  nrdefault, 43
  nrels, 43
  ntime, 43

phi, 43
print_level, 43
refine, 43
rfactors, 43
rnorm, 43
rtol, 44
sum, 44
tol, 44
tstart, 44
tstop, 44
write, 44
write_level, 44
_braid_CoreElt
_braid.h, 48
_braid_CoreFcn
_braid.h, 49
_braid_DestroyStatus
_braid.h, 50
_braid_FInterp
_braid.h, 50
_braid_FRefine
_braid.h, 50
_braid_FRestrict
_braid.h, 50
_braid_FWrite
_braid.h, 51
_braid_GetDistribution
_braid.h, 51
_braid_GetProc
_braid.h, 51
_braid_Grid, 44
cfactor, 45
clower, 45
cupper, 45
ilower, 45
iupper, 45
level, 45
ncpoints, 45
recv_handle, 45
recv_index, 45
send_handle, 45
send_index, 45
ta, 45
ta_alloc, 46
ua, 46
ua_alloc, 46
va, 46
va_alloc, 46
wa, 46
wa_alloc, 46
_braid_GridDestroy
_braid.h, 51
_braid_GridElt
_braid.h, 49
_braid_GridInit
_braid.h, 51
_braid_InitGuess
_braid.h, 51
_braid_InitHierarchy
_braid.h, 51
_braid_InitStatus
_braid.h, 51
_braid_IsCPoint
_braid.h, 49
_braid_IsFPoint
_braid.h, 49
_braid_MapCoarseToFine
_braid.h, 49
_braid_MapFineToCoarse
_braid.h, 49
_braid_ParFprintfFlush
util.h, 55
_braid_Phi
_braid.h, 51
_braid_ProjectInterval
util.h, 56
_braid_Refine
_braid.h, 51
_braid_SetAccuracy
util.h, 56
_braid_Status, 46
done, 46
iter, 46
level, 47
rnorm, 47
_braid_StatusElt
_braid.h, 49
_braid_Step
_braid.h, 52
_braid_TAlloc
_braid.h, 49
_braid_TFree
_braid.h, 49
_braid_TReAlloc
_braid.h, 49
_braid_UCommInit
_braid.h, 52
_braid_UCommInitF
_braid.h, 52
_braid_UCommWait
_braid.h, 52
_braid_UGetInterval
_braid.h, 52
_braid_UGetVector
_braid.h, 52
_braid_UGetVectorRef
_braid.h, 52
_braid_USetVector

_braid.h, 52
_braid_USetVectorRef
_braid.h, 52
_braid_UWriteVector
_braid.h, 53
_braid_error_flag
_braid.h, 53
_braid_printf
util.h, 56
_braid_printfile
_braid.h, 53

accuracy
_braid_Core, 41
app
_braid_Core, 41

Braid test routines, 35
braid_TestAll, 35
braid_TestBuf, 36
braid_TestClone, 36
braid_TestCoarsenRefine, 37
braid_TestDot, 37
braid_TestInitWrite, 38
braid_TestSum, 38
braid.h, 53
braid_Int, 54
braid_Real, 54
braid_App
User-written routines, 24
braid_Core
User interface routines, 26
braid_Destroy
User interface routines, 27
braid_Drive
User interface routines, 28
braid_GetNumIter
User interface routines, 28
braid_GetRNorm
User interface routines, 28
braid_GetStatusDone
User interface routines, 28
braid_GetStatusIter
User interface routines, 28
braid_GetStatusLevel
User interface routines, 28
braid_GetStatusResidual
User interface routines, 29
braid_Init
User interface routines, 29
braid_Int
braid.h, 54
braid_PrintStats
User interface routines, 29
braid_PtFcnBufPack
User-written routines, 24
braid_PtFcnBufSize
User-written routines, 24
braid_PtFcnBufUnpack
User-written routines, 24
braid_PtFcnClone
User-written routines, 24
braid_PtFcnCoarsen
User-written routines, 24
braid_PtFcnDot
User-written routines, 24
braid_PtFcnFree
User-written routines, 24
braid_PtFcnInit
User-written routines, 24
braid_PtFcnPhi
User-written routines, 25
braid_PtFcnRefine
User-written routines, 25
braid_PtFcnSum
User-written routines, 25
braid_PtFcnWrite
User-written routines, 25
braid_Real
braid.h, 54
braid_SetAbsTol
User interface routines, 29
braid_SetCFactor
User interface routines, 31
braid_SetFMG
User interface routines, 31
braid_SetLoosexTol
User interface routines, 31
braid_SetMaxCoarse
User interface routines, 31
braid_SetMaxIter
User interface routines, 31
braid_SetMaxLevels
User interface routines, 31
braid_SetNFMGVcyc
User interface routines, 32
braid_SetNRelax
User interface routines, 32
braid_SetPrintFile
User interface routines, 32
braid_SetPrintLevel
User interface routines, 32
braid_SetRelTol
User interface routines, 32
braid_SetSpatialCoarsen
User interface routines, 33
braid_SetSpatialRefine
User interface routines, 33

braid_SetTightxTol
    User interface routines, 33
braid_SetWriteLevel
    User interface routines, 33
braid_SplitCommworld
    User interface routines, 33
braid_Status
    User-written routines, 25
braid_TestAll
    Braid test routines, 35
braid_TestBuf
    Braid test routines, 36
braid_TestClone
    Braid test routines, 36
braid_TestCoarsenRefine
    Braid test routines, 37
braid_TestDot
    Braid test routines, 37
braid_TestInitWrite
    Braid test routines, 38
braid_TestSum
    Braid test routines, 38
braid_Vector
    User-written routines, 25
braid_test.h, 55
buffer
    _braid_CommHandle, 40
bufpack
    _braid_Core, 41
bufsize
    _braid_Core, 41
bufunpack
    _braid_Core, 42

cfactor
    _braid_Grid, 45
cfactors
    _braid_Core, 42
cfdefault
    _braid_Core, 42
clone
    _braid_Core, 42
clower
    _braid_Grid, 45
coarsen
    _braid_Core, 42
comm
    _braid_Core, 42
comm_world
    _braid_Core, 42
cupper
    _braid_Grid, 45

done
    _braid_Status, 46
dot
    _braid_Core, 42

fmg
    _braid_Core, 42
free
    _braid_Core, 42

globaltime
    _braid_Core, 42
grids
    _braid_Core, 42
gupper
    _braid_Core, 42

ilower
    _braid_Grid, 45
init
    _braid_Core, 42
iter
    _braid_Status, 46
iupper
    _braid_Grid, 45

level
    _braid_Grid, 45
    _braid_Status, 47
localtime
    _braid_Core, 42
loose
    _braid_AccuracyHandle, 39

matchF
    _braid_AccuracyHandle, 39
max_coarse
    _braid_Core, 43
max_iter
    _braid_Core, 43
max_levels
    _braid_Core, 43

ncpoints
    _braid_Grid, 45
nfmg_Vcyc
    _braid_Core, 43
niter
    _braid_Core, 43
nlevels
    _braid_Core, 43
nrdefault
    _braid_Core, 43
nrels
    _braid_Core, 43
ntime

_braid_Core, 43
num_requests
_braid_CommHandle, 40

old_value
_braid_AccuracyHandle, 39

phi
_braid_Core, 43
print_level
_braid_Core, 43

recv_handle
_braid_Grid, 45
recv_index
_braid_Grid, 45
refine
_braid_Core, 43
request_type
_braid_CommHandle, 40
requests
_braid_CommHandle, 40
rfactors
_braid_Core, 43
rnorm
_braid_Core, 43
_braid_Status, 47
rtol
_braid_Core, 44

send_handle
_braid_Grid, 45
send_index
_braid_Grid, 45
status
_braid_CommHandle, 40
sum
_braid_Core, 44

ta
_braid_Grid, 45
ta_alloc
_braid_Grid, 46
tight
_braid_AccuracyHandle, 39
tight_used
_braid_AccuracyHandle, 39
tol
_braid_Core, 44
tstart
_braid_Core, 44
tstop
_braid_Core, 44

ua
_braid_Grid, 46
ua_alloc
_braid_Grid, 46
User interface routines, 26
braid_Core, 26
braid_Destroy, 27
braid_Drive, 28
braid_GetNumIter, 28
braid_GetRNorm, 28
braid_GetStatusDone, 28
braid_GetStatusIter, 28
braid_GetStatusLevel, 28
braid_GetStatusResidual, 29
braid_Init, 29
braid_PrintStats, 29
braid_SetAbsTol, 29
braid_SetCFactor, 31
braid_SetFMG, 31
braid_SetLoosexTol, 31
braid_SetMaxCoarse, 31
braid_SetMaxIter, 31
braid_SetMaxLevels, 31
braid_SetNFMGVcyc, 32
braid_SetNRelax, 32
braid_SetPrintFile, 32
braid_SetPrintLevel, 32
braid_SetRelTol, 32
braid_SetSpatialCoarsen, 33
braid_SetSpatialRefine, 33
braid_SetTightxTol, 33
braid_SetWriteLevel, 33
braid_SplitCommworld, 33
User-written routines, 23
braid_App, 24
braid_PtFcnBufPack, 24
braid_PtFcnBufSize, 24
braid_PtFcnBufUnpack, 24
braid_PtFcnClone, 24
braid_PtFcnCoarsen, 24
braid_PtFcnDot, 24
braid_PtFcnFree, 24
braid_PtFcnInit, 24
braid_PtFcnPhi, 25
braid_PtFcnRefine, 25
braid_PtFcnSum, 25
braid_PtFcnWrite, 25
braid_Status, 25
braid_Vector, 25
util.h, 55
_braid_ParFprintfFlush, 55
_braid_ProjectInterval, 56
_braid_SetAccuracy, 56
_braid_printf, 56

va
_braid_Grid, 46
va_alloc
_braid_Grid, 46
value
_braid_AccuracyHandle, 39
vector_ptr
_braid_CommHandle, 40

wa
_braid_Grid, 46
wa_alloc
_braid_Grid, 46
write
_braid_Core, 44
write_level
_braid_Core, 44