XBraid Tutorial

A flexible and scalable approach to parallel-in-time

Rob Falgout and Jacob Schroder
Fifth Parallel-in-Time Integration Workshop
November 29th, 2016
Outline

1. Introduction
   → Tutorial software requirements and XBraid overview

2. Simplest example of solving a scalar ODE with examples/ex-01
   → Defining the App and vector structures, writing wrapper functions, running XBraid

3. Explore more XBraid settings in examples/ex-01-expanded.c

4. Explore advanced XBraid features
   → Adaptivity, residual and storage options, shell-vectors and BDF-k, spatial coarsening
   → Fortran90 Interface

5. Porting a user-code to XBraid with examples/ex-02
   → Debugging the connection to XBraid
   → Intrusiveness versus efficiency

6. A few application area highlights
To interact with the tutorial, you need

- This tutorial needs a working installation of XBraid 2.1 or higher
  http://llnl.gov/casc/tbraid
  • See the User’s manual for instructions on how to install XBraid

- XBraid v2.1 (or higher) required
- GCC compiler required
- MPI recommended
- Python 2.7 (or higher) with NumPy, Matplotlib recommended
- hypre installation for running ex-03 optional
  http://llnl.gov/casc/hypre
To interact with the tutorial, you need

- Make sure you can run

```bash
$ cd xbraid
$ make
$ cd examples
$ make ex-01 ex-02
$ ./ex-01
Braid: || r_1 || = 2.845538e-02, conv factor = 1.00e+00, wall time = ...
Braid: || r_2 || = 8.621939e-04, conv factor = 3.03e-02, wall time = ...
Braid: || r_3 || = 0.000000e+00, conv factor = 0.00e+00, wall time = ...
...
$ ./ex-02
Braid: || r_0 || = 4.041694e+00, conv factor = 1.00e+00, wall time = ...
Braid: || r_1 || = 1.037471e-01, conv factor = 2.57e-02, wall time = ...
Braid: || r_2 || = 2.926906e-03, conv factor = 2.82e-02, wall time = ...
...
```
Traditional time integration will become a sequential bottleneck

Clock rates are no longer increasing – faster speed is now achieved through more concurrency

Parallel time integration methods are needed (think exascale)!
Multigrid is well suited for exascale

- For many applications, the fastest and most scalable solvers are already multigrid methods

- Exascale solver algorithms will need to:
  - Exhibit extreme levels of parallelism (exascale $\rightarrow$ 1 billion cores)
    Spatial multigrid has already scaled to over 1 million cores
  - Minimize data movement
    Multigrid is $O(N)$ optimal
  - Exploit machine heterogeneity
    If the user’s problem can exploit heterogenity, then so can multigrid
  - Be resilient to faults
    Multigrid has already shown good resilience (being iterative and multilevel helps)

- Apply multigrid to the temporal dimension!
Our approach for parallel-in-time

- Apply the wealth of research on parallel spatial multigrid to multigrid in time
- This is where our team has extensive experience (hypre project)
Technical approach

- Consider the **general** one-step method
  \[ u_i = \Phi_i(u_{i-1}) + g_i, \quad i = 1, 2, \ldots, N \]

- In the linear setting *(for simplicity)*, time marching \( \equiv \) forward solve
  - This is an \( O(N) \) direct method, **but sequential**

\[
Au \equiv \begin{pmatrix}
I & & & & \\
-\Phi & I & & & \\
& \ddots & \ddots & & \\
& & -\Phi & I & \\
& & & & -\Phi & I
\end{pmatrix}
\begin{pmatrix}
u_0 \\
u_1 \\
\vdots \\
u_N \\
g_0 \\
g_1 \\
\vdots \\
g_N
\end{pmatrix} \equiv g
\]

- We propose solving this system **iteratively** with a multigrid method
  - Extend multigrid reduction (MGR, 1979) to the time dimension
  - Coarsens only in time (non-intrusive)
  - \( O(N) \), highly parallel
Technical approach

- Relaxation is highly parallel
  - Alternates between $F$-points and $C$-points
  - $F$-point relaxation = integration over each coarse time interval

- Coarse system is a time rediscretization
  - Approximate impractical $\Phi^m$ with $\Phi_\Delta$ a rediscretization with $\Delta T$

\[
A_\Delta = \begin{pmatrix}
I \\
-\Phi^m & I \\
\vdots & \ddots & \ddots \\
-\Phi^m & I
\end{pmatrix} \quad \Rightarrow \quad A_\Delta = \begin{pmatrix}
I \\
-\Phi_\Delta & I \\
\vdots & \ddots & \ddots \\
-\Phi_\Delta & I
\end{pmatrix}
\]

  - Apply recursively for multilevel hierarchy
Parallel decomposition

- Our code **XBraid** is agnostic to spatial decomposition and only parallelizes in time.

**Serial time stepping**
- Parallelize in space only
- Store only one time step

**Multigrid in time**
- Parallelize in space and time
- Store several time steps
Properties of the approach

- Expose **concurrency** in the time dimension with multigrid
- **Non-intrusive**, with unchanged time discretization
- Converges to **same solution** as sequential time stepping

- Only store **C-points** to minimize storage

- Optimal for variety of parabolic problems
  - Converges in ~10 iterations for any coarsening factor
  - Larger factors require larger (sequential) F-relaxation intervals

- Extends to **nonlinear** problems with FAS formulation

- In simple two-level setting, our method is **equivalent to parareal**
  - This is a popular method, typically not viewed as multigrid

- Many active research topics
  - Adaptivity in time, moving meshes and multistep methods all possible
  - Space-time coarsening possible (stability on coarse time-grids for explicit schemes)
Huge parallel speedups available, but in a new way

- Time stepping is already $O(N)$
- Useful only beyond a crossover
- Need 10-100x more parallelism just to break even

The more time steps, the more speedup potential
- Applications that require lots of time steps will benefit first
- Speedups (so far) up to 52x on 100K total cores

3D Heat Equation: $33^3 \times 4097$, 8 procs in space, 6x speedup
XBraid: open source, non-intrusive and flexible

- Overlap communication and computation
  - Consider relaxation over a processor’s portion of the time interval
  - Start computation with right-most interval to overlap comm/comp

1) Post receive
2) Compute and send
3) Compute other points, moving right to left

- Code stores only $C$-points to minimize storage
  - Ability to coarsen by large factors means fewer parallel resources
  - Memory multiplier per processor
    $\sim O(\log N)$ with time coarsening, $O(1)$ with space-time coarsening
XBraid: open source, non-intrusive and flexible

- User defines two objects:
  - App and Vector

- User also writes several wrapper routines:
  - Step, Init, Clone, Sum, SpatialNorm, Access, BufPack, BufUnpack, BufSize
  - For optional spatial coarsening: Coarsen, Refine

- Example: Step(app, u, status)
  - Advances vector $u$ from time $t_{start}$ to $t_{stop}$
  - Returns a target refinement factor
Outline

1. Introduction
   → Tutorial software requirements and XBraid overview

2. Simplest example of solving a scalar ODE with examples/ex-01
   → Defining the App and vector structures, writing wrapper functions, running XBraid

3. Explore more XBraid settings in examples/ex-01-expanded.c

4. Explore advanced XBraid features
   → Adaptivity, residual and storage options, shell-vectors and BDF-k, spatial coarsening
   → Fortran90 Interface

5. Porting a user-code to XBraid with examples/ex-02
   → Debugging the connection to XBraid
   → Intrusiveness versus efficiency

6. A few application area highlights
Simplest Example: Scalar ODE

- **File**: examples/ex-01.c  
  Solves: \( u_t = \lambda u \)
- First, you must define your **app** and **vector** structures

This is your simulation application structure. Place any time-independent data here, which is needed to take a time step.

Here, we only need the MPI rank in the App structure (for later file output).

```c
typedef struct _braid_App_struct{
    int       rank;
} my_App;

typedef struct _braid_Vector_struct{
    double value;
} my_Vector;
```
Simplest Example: Scalar ODE

- **File:** examples/ex-01.c
- **Solves:** \( u_t = \lambda u \)
- **First, you must define your app and vector structures**

This is your state vector structure. It holds any time-dependent information that should stay with a vector, e.g. mesh information and unknowns.

For this problem, the vector is a single double.

```c
typedef struct _braid_App_struct{
    int rank;
} my_App;

typedef struct _braid_Vector_struct{
    double value;
} my_Vector;
```
Define the \texttt{Step()} function

- \textbf{File:} examples/ex-01.c \hspace{1cm} \textbf{Solves:} \( u_t = \lambda u \)

\texttt{Step()} evolves \( u \) from \( t_{\text{start}} \) to \( t_{\text{stop}} \)

```c
int my_Step(braid_App app,
             braid_Vector ustop,
             braid_Vector fstop,
             braid_Vector u,
             braid_StepStatus status)
{
    double tstart;
    double tstop;
    braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

    (u->value) = 1./(1. + tstop-tstart)*(u->value);

    return 0;
}
```
Define the \texttt{Step()} function

- File: \texttt{examples/ex-01.c}  \quad Solves: $u_t = \lambda u$

The app structure is passed into every user-written function.

```
int my_Step(braid_App app,
            braid_Vector ustop,
            braid_Vector fstop,
            braid_Vector u,
            braid_StepStatus status)
{
    double tstart;
    double tstop;
    braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

    (u->value) = 1./(1. + tstop-tstart)*(u->value);

    return 0;
}
```
Define the **Step()** function

- **File:** `examples/ex-01.c`  
  **Solves:** $u_t = \lambda u$

```c
int my_Step(braid_App app,
            braid_Vector ustop,
            braid_Vector fstop,
            braid_Vector u,
            braid_StepStatus status)
{
  double tstart;
  double tstop;
  braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

  (u->value) = 1./(1. + tstop-tstart)*(u->value);

  return 0;
}
```
Define the \texttt{Step()} function

- File: \texttt{examples/ex-01.c}  
  Solves: $u_t = \lambda u$

```c
int my_Step(braid_App app,
             braid_Vector ustop,
             braid_Vector fstop,
             braid_Vector u,
             braid_StepStatus status)
{
    double tstart;
    double tstop;
    braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

    (u->value) = 1./(1. + tstop-tstart)*(u->value);

    return 0;
}
```

Vector at \texttt{tstart}
Define the `Step()` function

- **File:** examples/ex-01.c  
  **Solves:** \( u_t = \lambda u \)

```c
int my_STEP(braid_APP app,
            braid_VECTOR ustop,
            braid_VECTOR fstop,
            braid_VECTOR u,
            braid_STEP_STATUS status)
{
    double tstart;
    double tstop;
    braid_STEP_STATUS_GET_TSTART_TSTOP(status, &tstart, &tstop);

    (u->value) = 1.0 / (1.0 + tstop - tstart) * (u->value);

    return 0;
}
```

Ignore by default. (XBraid forcing term, only needed if residual option is used)
Define the \texttt{Step()} function

- **File:** examples/ex-01.c  
  **Solves:** $u_t = \lambda u$

```
int my_Step(braid_App app,
    braid_Vector ustop,
    braid_Vector fstop,
    braid_Vector u,
    braid_StepStatus status)
{
    double tstart;
    double tstop;
    braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

    (u->value) = 1./(1. + tstop-tstart)*(u->value);

    return 0;
}
```

Status structures can be queried for various information (level, iteration, etc...).
Define the `Step()` function

- **File:** examples/ex-01.c  
  **Solves:** $u_t = \lambda u$

For instance, to get `tstart`, `tstop`

```c
int my_Step(braid_App app,
            braid_Vector ustop,
            braid_Vector fstop,
            braid_Vector u,
            braid_StepStatus status)
{
    double tstart;
    double tstop;
    braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

    (u->value) = 1./(1. + tstop-tstart)*(u->value);

    return 0;
}
```
Define the `Step()` function

- File: `examples/ex-01.c`  
Solves: $u_t = \lambda u$

### Take backward Euler step

```c
int my_Step(braid_App app,
             braid_Vector ustop,
             braid_Vector fstop,
             braid_Vector u,
             braid_StepStatus status)
{
    double tstart;
    double tstop;
    braid_StepStatusGetTstartTstop(status, &tstart, &tstop);

    (u->value) = 1./(1. + tstop-tstart)*(u->value);

    return 0;
}
```
Define other wrapper functions

- File: examples/ex-01.c  
  Solves: $u_t = \lambda u$
- Define functions: Init, Clone, Free, Sum, SpatialNorm, Access, BufPack, BufUnpack, BufSize

Again, we see the app structure being passed in

```c
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;
}
```
Define other wrapper functions

- File: examples/ex-01.c  
  Solves: $u_t = \lambda u$

- Define functions: Init, Clone, Free, Sum, SpatialNorm, Access, BufPack, BufUnpack, BufSize

This function carries out a simply AXPY operation.

```c
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;
}
```
Define other wrapper functions

- File: examples/ex-01.c  
  Solves: $u_t = \lambda u$

- Define functions: Init, Clone, Free, Sum, SpatialNorm, Access, BufPack, BufUnpack, BufSize

This function is how the user accesses the solution.
- By default, it is called at the end of the simulation for every time point.
- Using `braid_AccessSetLevel()` allows for more frequent access.

```c
int my_Access(braid_App app, braid_Vector u, braid_AccessStatus astatus)
{
    int index; char filename[255]; FILE *file;

    braid_AccessStatusGetTIndex(astatus, &index);
    sprintf(filename, "%s.%04d.%03d", "ex-01.out", index, app->rank);
    file = fopen(filename, "w");
    fprintf(file, "%.14e\n", (u->value));
    fflush(file); fclose(file); return 0;
} 
```
Define other wrapper functions

- **File:** examples/ex-01.c  
  **Solves:** $u_t = \lambda u$

- **Define functions:** Init, Clone, Free, Sum, SpatialNorm, Access, BufPack, BufUnpack, BufSize

Here, we just write a single solution value to individual files.

```c
int my_Access(braid_App app,
              braid_Vector u,
              braid_AccessStatus astatus)
{
    int index;  char filename[255];  FILE  *file;
    braid_AccessStatusGetTIndex(astatus, &index);
    sprintf(filename, "%s.%04d.%03d", "ex-01.out", index, app->rank);
    file = fopen(filename, "w");
    fprintf(file, "%.14e\n", (u->value));
    fflush(file);  fclose(file); return 0;
}
```
Define other wrapper functions

- **File**: examples/ex-01.c  
  **Solves**: \( u_t = \lambda u \)

- **Define functions**: Init, Clone, Free, Sum, SpatialNorm, Access, **BufPack**, BufUnpack, BufSize

The Buf* functions tell XBraid how to pack, unpack and size MPI Buffers.
Define other wrapper functions

- **File:** examples/ex-01.c  
  Solves: \( u_t = \lambda u \)

- **Define functions:** Init, Clone, Free, Sum, SpatialNorm, Access, **BufPack**, BufUnpack, BufSize

```
int my_BufPack(braid_App app, braid_Vector u, void *buffer, braid_BufferStatus bstatus)
{
    double *dbuffer = buffer;
    dbuffer[0] = (u->value);
    braid_BufferStatusSetSize( bstatus, sizeof(double) );
    return 0;
}
```

**BufPack() flattens the vector** \( u \) **into** \( \text{buffer} \)

\( u_t = \lambda u \)
Define other wrapper functions

- **File:** examples/ex-01.c  
  **Solves:** $u_t = \lambda u$

- **Define functions:** Init, Clone, Free, Sum, SpatialNorm, Access, **BufPack**, BufUnpack, BufSize

Packing this buffer entails just setting a single double value.

```c
int my_BufPack(braid_App app, braid_Vector u, void *buffer, braid_BufferStatus bstatus)
{
    double *dbuffer = buffer;
    dbuffer[0] = (u->value);
    braid_BufferStatusSetSize( bstatus, sizeof(double) );
    return 0;
}
```
Define other wrapper functions

- File: examples/ex-01.c  
  Solves: $u_t = \lambda u$
- Define functions: Init, Clone, Free, Sum, SpatialNorm, Access, **BufPack**, BufUnpack, BufSize

This is an example of returning a value with a status structure, the buffer size:

```c
int my_BufPack(braid_App app, braid_Vector u, void *buffer, braid_BufferStatus bstatus)
{
    double *dbuffer = buffer;
    dbuffer[0] = (u->value);
    braid_BufferStatusSetSize( bstatus, sizeof(double) );
    return 0;
}
```
Initialize App and XBraid

- File: examples/ex-01.c  Solves: $u_t = \lambda u$
- The next step is to setup XBraid in `main()`

```c
int main()
{
    ...
    braid_Core    core;
    ntime  = 10;
    tstart = 0.0; tstop  = 5.0;
    ...
    app = (my_App *) malloc(sizeof(my_App));
    app->rank)   = rank;
    ...
    braid_Init(MPI_COMM_WORLD, MPI_COMM_WORLD, tstart, tstop,
               ntime, app, my_Step, my_Init, my_Clone,
               my_Free, my_Sum, my_SpatialNorm,
               my_Access, my_BufSize, my_BufPack,
               my_BufUnpack, &core);
}```
Initialize App and XBraid

- File: examples/ex-01.c
- Solves: \( u_t = \lambda u \)
- The next step is to setup XBraid in \( \text{main()} \)

braid_Core is the core data structure, holding all of XBraid's internals

```c
int main()
...
    braid_Core core;
    ntime = 10;
    tstart = 0.0; tstop = 5.0;
...
    app = (my_App *) malloc(sizeof(my_App));
    (app->rank) = rank;
...
    braid_Init(MPI_COMM_WORLD, MPI_COMM_WORLD, tstart, tstop,
               ntime, app, my_Step, my_Init, my_Clone,
               my_Free, my_Sum, my_SpatialNorm,
               my_Access, my_BufSize, my_BufPack,
               my_BufUnpack, &core);
```
Initialize App and XBraid

- File: examples/ex-01.c  
  Solves: $u_t = \lambda u$
- The next step is to setup XBraid in `main()`

Define your time domain

```c
int main()
{
  ...
  braid_Core    core;
  ntime  = 10;
  tstart = 0.0; tstop  = 5.0;
  ...
  app = (my_App *) malloc(sizeof(my_App));
  (app->rank)   = rank;
  ...
  braid_Init(MPI_COMM_WORLD, MPI_COMM_WORLD, tstart, tstop,
             ntime, app, my_Step, my_Init, my_Clone,
             my_Free, my_Sum, my_SpatialNorm,
             my_Access, my_BufSize, my_BufPack,
             my_BufUnpack, &core);
  ...
}
```
Initialize App and XBraid

- **File:** examples/ex-01.c   **Solves:** \( u_t = \lambda u \)
- **The next step is to setup XBraid in** `main()`

```c
int main()
...
    braid_Core    core;
    ntime  = 10;
    tstart = 0.0; tstop  = 5.0;
...
    app = (my_App *) malloc(sizeof(my_App));
    (app->rank)   = rank;
...
    braid_Init(MPI_COMM_WORLD, MPI_COMM_WORLD, tstart, tstop,
               ntime, app, my_Step, my_Init, my_Clone,
               my_Free, my_Sum, my_SpatialNorm,
               my_Access, my_BufSize, my_BufPack,
               my_BufUnpack, &core);
```
Initialize App and XBraid

- File: examples/ex-01.c
- Solves: $u_t = \lambda u$
- The next step is to setup `App` and `braid_Core` in `main()`

```
int main()
...
    braid_Core core;
    ntime = 10;
    tstart = 0.0; tstop = 5.0;
...
    app = (my_App *) malloc(sizeof(my_App));
    (app->rank) = rank;
...
    braid_Init(MPI_COMM_WORLD, MPI_COMM_WORLD, tstart, tstop,
               ntime, app, my_Step, my_Init, my_Clone,
               my_Free, my_Sum, my_SpatialNorm,
               my_Access, my_BufSize, my_BufPack,
               my_BufUnpack, &core);
```
Set XBraid options and run

- **File:** examples/ex-01.c  
  **Solves:** $u_t = \lambda u$
- **The next step is to setup** App and braid_Core in main()

```c
int main()
{
  ...
  braid_SetPrintLevel( core, 1);
  braid_SetMaxLevels(core, 2);
  braid_SetAbsTol(core, 1.0e-06);
  braid_SetCFactor(core, -1, 2);
  braid_Drive(core);
  braid_Destroy(core);
}
```
Set XBraid options and run

- **File:** examples/ex-01.c  
  **Solves:** \( u_t = \lambda u \)
- The next step is to setup App and braid-Core in `main()`

```c
int main()
{
  ...  
  braid_SetPrintLevel( core, 1);
  braid_SetMaxLevels(core, 2);
  braid_SetAbsTol(core, 1.0e-06);
  braid_SetCFactor(core, -1, 2);
  braid_Drive(core);
  braid_Destroy(core);
}
```
Set XBraid options and run

- **File**: examples/ex-01.c  
  **Solves**: $u_t = \lambda u$

- The next step is to setup App and braid_Core in **main()**

```c
int main()
    ...
    braid_SetPrintLevel( core, 1);
    braid_SetMaxLevels(core, 2);
    braid_SetAbsTol(core, 1.0e-06);
    braid_SetCFactor(core, -1, 2);

    braid_Drive(core);

    braid_Destroy(core);
```
Output

- **File:** examples/ex-01.c
- **Finally! We can run the example.**

```
$ cd examples
$ make ex-01
$ ./ex-01
$ cat ex-01.out.00*
  1.00000000000000e+00
  6.66666666666667e-01
  4.44444444444444e-01
  2.96296296296296e-01
  1.97530864197531e-01
  1.31687242798354e-01
  8.77914951989026e-02
  5.85276634659351e-02
  3.90184423106234e-02
  2.60122948737489e-02
  1.73415299158326e-02
```

Solves: \( u_t = \lambda u \)
Outline

1. Introduction
   → Tutorial software requirements and XBraid overview

2. Simplest example of solving a scalar ODE with examples/ex-01
   → Defining the App and vector structures, writing wrapper functions, running XBraid

3. Explore more XBraid settings in examples/ex-01-expanded.c

4. Explore advanced XBraid features
   → Adaptivity, residual and storage options, shell-vectors and BDF-k, spatial coarsening
   → Fortran90 Interface

5. Porting a user-code to XBraid with examples/ex-02
   → Debugging the connection to XBraid
   → Intrusiveness versus efficiency

6. A few application area highlights
Moving to `ex-01-expanded.c`

- **File:** `examples/ex-01-expanded.c`  
  **Solves:** $u_t = \lambda u$
- Adds more XBraid features and a command line interface to `ex-01.c`

Let's experiment with these options!

```plaintext
$ cd examples
$ make ex-01-expanded
$ ./ex-01-expanded -help
```

- `-ntime <ntime>` : set num time points
- `-ml <max_levels>` : set max levels
- `-nu <nrelax>` : set num F-C relaxations
- `-nu0 <nrelax>` : set num F-C relaxations on level 0
- `-tol <tol>` : set stopping tolerance
- `-cf <cfactor>` : set coarsening factor
- `-mi <max_iter>` : set max iterations
- `-fmg` : use FMG cycling
- `-res` : use my residual
- `-tg <mydt>` : use user-specified time grid
  - 1 - uniform time grid
  - 2 - nonuniform time grid
Examine the standard XBraid output

- File: examples/ex-01-expanded.c
  Solves: \( u_t = \lambda u \)

Residual history is printed out, along with convergence factors and wall times

```
$ ./ex-01-expanded
Braid: Begin simulation, 10 time steps
Braid: \| r_0 \| not available, wall time = 1.81e-04
Braid: \| r_1 \| = 2.845538e-02, conv factor = 1.00e+00, wall time = ...
Braid: \| r_2 \| = 8.621939e-04, conv factor = 3.03e-02, wall time = ...
Braid: \| r_3 \| = 0.000000e+00, conv factor = 0.00e+00, wall time = ...

start time = 0.000000e+00
stop time  = 5.000000e+00
time steps = 10

use seq soln? = 0
storage = -1

stopping tolerance = 1.000000e-06
use relative tol? = 0
max iterations = 100
iterations = 4
residual norm = 0.000000e+00

--> 2-norm TemporalNorm
```
Examine the standard XBraid output

- File: examples/ex-01-expanded.c  Solves: $u_t = \lambda u$

**Basic time domain information**

```plaintext
$ ./ex-01-expanded
  Braid: Begin simulation, 10 time steps
  Braid: || r_0 || not available, wall time = 1.81e-04
  Braid: || r_1 || = 2.845538e-02, conv factor = 1.00e+00, wall time = ...
  Braid: || r_2 || = 8.621939e-04, conv factor = 3.03e-02, wall time = ...
  Braid: || r_3 || = 0.000000e+00, conv factor = 0.00e+00, wall time = ...

  start time = 0.000000e+00
  stop time  = 5.000000e+00
  time steps = 10

  use seq soln? = 0
  storage      = -1

  stopping tolerance = 1.000000e-06
  use relative tol? = 0
  max iterations   = 100
  iterations       = 4
  residual norm    = 0.000000e+00
                     --> 2-norm TemporalNorm
```
Examine the standard XBraid output

- **File:** examples/ex-01-expanded.c  **Solves:** $u_t = \lambda u$

```bash
$ ./ex-01-expanded
Braid: Begin simulation, 10 time steps
Braid: || r_0 || not available, wall time = 1.81e-04
Braid: || r_1 || = 2.845538e-02, conv factor = 1.00e+00, wall time = ...
Braid: || r_2 || = 8.621939e-04, conv factor = 3.03e-02, wall time = ...
Braid: || r_3 || = 0.000000e+00, conv factor = 0.00e+00, wall time = ...

start time = 0.000000e+00
stop time  = 5.000000e+00
time steps = 10

use seq soln?         = 0
storage               = -1

stopping tolerance    = 1.000000e-06
use relative tol?     = 0
max iterations        = 100
iterations            = 4
residual norm         = 0.000000e+00
 ---> 2-norm TemporalNorm
```
Examine the standard XBraid output

- File: examples/ex-01-expanded.c  Solves: $u_t = \lambda u$

Describe the XBraid options set for this run

```
$ ./ex-01-expanded
  Braid: Begin simulation, 10 time steps
  Braid: || r_0 || not available, wall time = ...
  Braid: || r_1 || = 2.845538e-02, conv factor = 1.00e+00, wall time = ...
  Braid: || r_2 || = 8.621939e-04, conv factor = 3.03e-02, wall time = ...
  Braid: || r_3 || = 0.000000e+00, conv factor = 0.00e+00, wall time = ...

  start time = 0.000000e+00
  stop time  = 5.000000e+00
  time steps = 10

  use seq soln?         = 0
  storage               = -1

  stopping tolerance    = 1.000000e-06
  use relative tol?     = 0
  max iterations        = 100
  iterations            = 4
  residual norm         = 0.000000e+00

  --> 2-norm TemporalNorm
```
Examine the standard XBraid output

- File: examples/ex-01-expanded.c  Solves: $u_t = \lambda u$

Describe the XBraid options set for this run

```
$ ./ex-01-expanded
  Braid: Begin simulation, 10 time steps
...

use fmg? = 0
access_level = 1
print_level = 1

max number of levels = 2
min coarse = 2
number of levels = 2
skip down cycle = 1
number of refinements = 0

level   time-pts   cfactor   nrelax
  0       10       2       1
  1        5

wall time = ...
```
Examine the standard XBraid output

- **File:** examples/ex-01-expanded.c  
  **Solves:** $u_t = \lambda u$

**Describes the levels in the XBraid hierarchy**

```plaintext
$ ./ex-01-expanded
  Braid: Begin simulation, 10 time steps
  ...

  use fmg? = 0
  access_level = 1
  print_level = 1

  max number of levels = 2
  min coarse = 2
  number of levels = 2
  skip down cycle = 1
  number of refinements = 0

  level  time-pts  cfactor  nrelax
      0       10      2       1
      1        5

  wall time = ...
```
Increase number of time points

- File: examples/ex-01-expanded.c  Solves: $u_t = \lambda u$

Now, compare the effects of increasing the time domain size

```
$ ./ex-01-expanded -ntime 16
  Braid: Begin simulation, 16 time steps
  Braid: || r_0 || not available, wall time = ...
  Braid: || r_1 || = 2.851025e-02, conv factor = 1.00e+00, wall time = ...
  Braid: || r_2 || = 1.040035e-03, conv factor = 3.65e-02, wall time = ...
  Braid: || r_3 || = 1.040035e-03, conv factor = 3.65e-02, wall time = ...
  Braid: || r_4 || = 3.530338e-05, conv factor = 3.39e-02, wall time = ...
  ...

$ ./ex-01-expanded -ntime 128
  Braid: Begin simulation, 128 time steps
  Braid: || r_0 || not available, wall time = ...
  Braid: || r_1 || = 2.851112e-02, conv factor = 1.00e+00, wall time = ...
  Braid: || r_2 || = 1.040035e-03, conv factor = 3.68e-02, wall time = ...
  Braid: || r_3 || = 3.530338e-05, conv factor = 3.39e-02, wall time = ...
  Braid: || r_4 || = 3.716892e-07, conv factor = 1.05e-02, wall time = ...
  ...
```
FCF-relaxation

- File: examples/ex-01-expanded.c  Solves: $u_t = \lambda u$

Observe how changing the number of FCF-relaxations improves convergence

```
$ ./ex-01-expanded -ntime 128 -nu 0
  Braid: Begin simulation, 128 time steps
  Braid: \| r_0 \| not available, wall time = ...
  Braid: \| r_1 \| = 6.415003e-02, conv factor = 1.00e+00, wall time = ...
  Braid: \| r_2 \| = 5.312734e-03, conv factor = 8.28e-02, wall time = ...
  Braid: \| r_3 \| = 5.055060e-04, conv factor = 9.51e-02, wall time = ...
  Braid: \| r_4 \| = 5.101391e-05, conv factor = 9.51e-02, wall time = ...
  Braid: \| r_5 \| = 5.290607e-06, conv factor = 1.04e-01, wall time = ...
  Braid: \| r_6 \| = 5.570496e-07, conv factor = 1.05e-01, wall time = ...

$ ./ex-01-expanded -ntime 128 -nu 3
  Braid: Begin simulation, 128 time steps
  Braid: \| r_0 \| not available, wall time = ...
  Braid: \| r_1 \| = 6.31827e-03, conv factor = 1.00e+00, wall time = ...
  Braid: \| r_2 \| = 4.094709e-05, conv factor = 7.27e-03, wall time = ...
  Braid: \| r_3 \| = 3.420453e-07, conv factor = 8.35e-03, wall time = ...
```
Halting tolerance and max-iterations

- **File:** examples/ex-01-expanded.c  **Solves:** $u_t = \lambda u$

Observe how changing the tolerance and max-iter (-mi) parameters affect XBraid

- $./ex-01-expanded -ntime 128 -tol 1e-3
  - iterations = 4

- $./ex-01-expanded -ntime 128 -tol 1e-12
  - iterations = 10

- $./ex-01-expanded -ntime 128 -tol 1e-12 -mi 3
  - iterations = 3
Full multigrid cycles (FMG)

- **File**: examples/ex-01-expanded.c  
  **Solves**: $u_t = \lambda u$

Now, use the fmg parameter and plot braid.out.cycle (file generated at runtime)

```
$ ./ex-01-expanded -ntime 32 -ml 15 -mi 4 -fmg
$ python ../user_utils/cycleplot.py
```

![XBraid Cycling](image)

This functionality can be used to adaptively refine in time (nested iteration)
Outline

1. Introduction
   → Tutorial software requirements and XBraid overview

2. Simplest example of solving a scalar ODE with examples/ex-01
   → Defining the App and vector structures, writing wrapper functions, running XBraid

3. Explore more XBraid settings in examples/ex-01-expanded.c

4. Explore advanced XBraid features
   → Adaption, residual and storage options, shell-vectors and BDF-k, spatial coarsening
   → Fortran90 Interface

5. Porting a user-code to XBraid with examples/ex-02
   → Debugging the connection to XBraid
   → Intrusiveness versus efficiency

6. A few application area highlights
Advanced feature: FMG allows for adaptivity in time and space

- User returns refinement factor in `Step()`
- Example time grid hierarchy

```
Level 0
Level 1
Level 2
```

- User requests refinement factors on the finest grid which generates a new grid and hierarchy

```
Level -1
Level 0
Level 1
Level 2
```

Notice new C-pts
Advanced feature: adaptivity in time

- **File:** examples/ex-03.c
  
  Solves: \( u_t = -u_{xx} - u_{yy} \)

- This simple example carries out naive pre-specified refinements
- \texttt{braid\_StepStatusSetRFactor(status, k)} refines an interval \( k \) times
  - Called from inside of \texttt{Step()}

$ \text{make ex-03}$

$ \text{./ex-03 -nt 128 -nx 9 9 -mi 4 -refine}$

Braid: Begin simulation, 128 time steps

Braid: \( || r_0 || \) not available, wall time = ...

Braid: \( || r_1 || = 5.002967e-01 \), conv factor = 1.00e+00, wall time = ...

Braid: Temporal refinement occurred, 242 time steps

Braid: \( || r_1 || = 2.810253e-02 \), conv factor = 1.00e+00, wall time = ...

Braid: Temporal refinement occurred, 390 time steps

Braid: \( || r_1 || = 3.136143e-03 \), conv factor = 1.00e+00, wall time = ...

Braid: Temporal refinement occurred, 583 time steps

Braid: \( || r_1 || = 1.197026e-03 \), conv factor = 1.00e+00, wall time = ...

Braid: \( || r_2 || = 1.558192e-04 \), conv factor = 1.30e-01, wall time = ...

Braid: \( || r_3 || = 1.623626e-05 \), conv factor = 1.04e-01, wall time = ...
Advanced feature: adaptivity in time

- File: examples/ex-03.c
- Solves: $u_t = -u_{xx} - u_{yy}$
- Now, visualize the cycling
- Observe how the new levels (and time-points) are added
- This causes an uneven reduction in the residual

$\text{Python code:} \\
\text{\$ python ../user_utils/cycleplot.py}$

Refinement here is with a V-cycle. But can also be done with FMG cycles.
Advanced feature: residual function

- File: examples/ex-01-expanded.c  Solves: $u_t = \lambda u$

Observe how turning on the residual function changes convergence

```
$./ex-01-expanded -ntime 128 -res 
... 
iterations = 7 

$./ex-01-expanded -ntime 128 
... 
iterations = 6 
```

- File: examples/ex-03.c  Solves: $u_t = -u_{xx} - u_{yy}$

```
$ make ex-03 
$ ./ex-03 -res -nt 128 -nx 9 9 -mi 4 
Braid: || r_1 || = 5.231464e-01, conv factor = 1.00e+00, wall time = ...
Braid: || r_2 || = 6.067546e-02, conv factor = 1.16e-01, wall time = ...

$ ./ex-03 -nt 128 -nx 9 9 -mi 4 
Braid: || r_1 || = 5.002967e-01, conv factor = 1.00e+00, wall time = ...
Braid: || r_2 || = 2.701758e-02, conv factor = 5.40e-02, wall time = ...
```
Understanding the residual feature

- XBraid computes the FAS residual in a block-row fashion for the space-time system

\[ A_i(u_i, u_{i-1}) = f_i \]

- Consider, for example, the common additive form of a user residual:

  User specifies this
  \[ A_i(u_i, u_{i-1}) = -\Phi(u_{i-1}) + \Psi(u_i) \]

  FAS residual computed internally
  \[ r_i = f_i + \Phi(u_{i-1}) - \Psi(u_i) \]

- **Default setting:** \( \text{Step}() = \Phi(u_i) \) and \( \Psi = I \)
  - XBraid can compute the rest of the residual on its own

- **Residual setting:** user defines a new function \( \text{Residual}(u_i, u_{i-1}) = A_i(u_i, u_{i-1}) \)
  - This function defines the equation to be solved, implying that \( \text{Step}() \) must be compatible.
  - \( \text{Step}() \) must now compute \( u_i = \Psi^{-1}(f_i + \Phi(u_{i-1})) \)
  - Notice how \( \text{Step}() \) must now account for \( f_i \), that is, \( f_{\text{stop}} \) in \( \text{Step}() \) is no longer NULL!

- **Computational savings:** consider the heat equation and backward Euler
  - **Default:** \( \text{Step}() \) implements \( \Phi \), a full implicit solve for an accurate residual
  - **Residual:** \( \text{Step}() \) implements a very weak inexact solve (only used for relaxation)
    \( \text{Residual}() \) uses \( \Phi = I \) and \( \Psi \) is just a sparse matrix (very cheap!)
Advanced feature: shell-vectors & BDF-k

- **File**: examples/ex-01-expanded-bdf2.c  **Solves**: $u_t = \lambda u$

- XBraid is designed for one-step methods, so we make BDF-k “one-step” by grouping $k$ time-steps together

  - Creates non-uniform time-step sizes on coarse grids

- The shell-vector feature allows for the storage of meta-data at every time point, including F-points that are otherwise not stored.
  - This meta-data allows for tracking the irregular time-grid spacing

- Other BDF-k strategies, like reducing order on coarse-grids, are possible

- To use the shell option, you must define new shell functions for allocating, copying, and freeing vector shells
Advanced feature: extra storage

- File: examples/ex-03.c
  Solves: \( u_t = -u_{xx} - u_{yy} \)

- Set a storage value \( k \) (default is -1):
  - For level \( \geq k \geq 0 \), store all points; for level \( < k \), store only C-points
  - \( k = 0 \) storage at all points on all levels
  - \( k = -1 \) special value, storage only at C-points on all levels

- The extra storage critically gives improved initial guesses to implicit solvers
- The extra storage changes the problem being solved
  - The operator \( \Phi \) changes as the initial guess changes

- Look at the residual histories with

\[
\begin{pmatrix}
  I & I \\
  -\Phi & I \\
  \vdots & \vdots \\
  -\Phi & I
\end{pmatrix}
\begin{pmatrix}
  u_0 \\
  u_1 \\
  \vdots \\
  u_N
\end{pmatrix} =
\begin{pmatrix}
  g_0 \\
  g_1 \\
  \vdots \\
  g_N
\end{pmatrix}
\]

- \( F \)-point (fine grid only)
- C-point (coarse & fine grid)
Advanced feature: skip option

- File: examples/ex-03.c  
  Solves: $u_t = -u_{xx} - u_{yy}$

- Skip allows XBraid to skip (typically useless) work on the first down cycle
  - By default, skip is turned on
  - This relaxation is typically useless
- Compare the residual histories for

  $\$/ex-03 -nx 17 17 -nt 128 -skip 1
  $\$/ex-03 -nx 17 17 -nt 128 -skip 0

---

![No Skip](image_url)

![Skip](image_url)
Advanced feature: parallel-run

- File: examples/ex-03.c  
  Solves: $u_t = -u_{xx} - u_{yy}$

Run in parallel!

```
$./mpirun -np 8 ex-03 -pgrid 2 2 2 -nt 256 -nx 17 17
Braid: || r_0 || not available, wall time = ...
Braid: || r_1 || = 6.166798e-01, conv factor = 1.00e+00, wall time = ...
Braid: || r_2 || = 2.319985e-02, conv factor = 3.76e-02, wall time = ...
Braid: || r_3 || = 6.972052e-04, conv factor = 3.01e-02, wall time = ...
Braid: || r_4 || = 1.135286e-05, conv factor = 1.63e-02, wall time = ...
...```
Advanced feature: spatial coarsening

- **File:** examples/ex-02.c
- **Solves:** $u_t = -u_{xx}$

Here, we use simple bilinear interpolation (and its transpose) for spatial coarsening.

```bash
.$/ex-02 -ntime 64 -nspace 17 -ml 3 -sc
Braid: || r_0 || = 2.935397e+00, conv factor = 1.00e+00, wall time = ...
Braid: || r_1 || = 1.483600e-01, conv factor = 5.05e-02, wall time = ...
Braid: || r_2 || = 3.884625e-03, conv factor = 2.62e-02, wall time = ...
Braid: || r_3 || = 1.315185e-04, conv factor = 3.39e-02, wall time = ...
...
```

<table>
<thead>
<tr>
<th>level</th>
<th>dx</th>
<th>dt</th>
<th>dt/dx^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.96e-01</td>
<td>9.82e-02</td>
<td>2.55e00</td>
</tr>
<tr>
<td>1</td>
<td>3.93e-01</td>
<td>1.96e-01</td>
<td>1.27e00</td>
</tr>
<tr>
<td>2</td>
<td>7.85e-01</td>
<td>3.93e-01</td>
<td>6.37e-01</td>
</tr>
</tbody>
</table>

```bash
.$/ex-02 -ntime 64 -nspace 17 -ml 3
Braid: || r_0 || = 2.935397e+00, conv factor = 1.00e+00, wall time = ...
Braid: || r_1 || = 1.666814e-01, conv factor = 5.68e-02, wall time = ...
Braid: || r_2 || = 8.328760e-03, conv factor = 5.00e-02, wall time = ...
Braid: || r_3 || = 2.844685e-04, conv factor = 3.42e-02, wall time = ...
...
```

<table>
<thead>
<tr>
<th>level</th>
<th>dx</th>
<th>dt</th>
<th>dt/dx^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.96e-01</td>
<td>9.82e-02</td>
<td>2.55e00</td>
</tr>
<tr>
<td>1</td>
<td>1.96e-01</td>
<td>1.96e-01</td>
<td>5.09e00</td>
</tr>
<tr>
<td>2</td>
<td>1.96e-01</td>
<td>3.93e-01</td>
<td>1.02e01</td>
</tr>
</tbody>
</table>

Spatial coarsening is active research and can (sometimes) negatively impact convergence.
Advanced feature: coarsening factor

- File: `examples/ex-02.c`  
  Solves: $u_t = -u_{xx}$

- Changing the coarsening factor does not change convergence (much)
- This powerful fact applies to parabolic problems in general
  - Allows for a great deal of performance tuning
  - Requires that FCF-relaxation or F-cycles be used

```
$./ex-02 -ntime 1024 -nspace 128 -cf 16 -ml 10
...
iterations = 7

$./ex-02 -ntime 1024 -nspace 128 -cf 2 -ml 10
...
iterations = 8
```
Fortran90 interface

- **File:** examples/ex-01-expanded-f.f90   **Solves:** $u_t = \lambda u$

**Uses Fortran90 modules to define the App and Vector Types**

```fortran
module braid_types

    type my_vector
        double precision val
    end type my_vector
...
```

**User-defined wrapper functions are the same, only written in Fortran90**

```fortran
subroutine braid_Sum_F90(app, alpha, x, beta, y)
    ! Braid types
    use braid_types
    implicit none
    type(my_vector) :: x, y
    type(my_app) :: app

    double precision alpha, beta
    y%val = alpha*(x%val) + beta*(y%val)
end subroutine braid_Sum_F90
```
Outline

1. Introduction
   → Tutorial software requirements and XBraid overview

2. Simplest example of solving a scalar ODE with examples/ex-01
   → Defining the App and vector structures, writing wrapper functions, running XBraid

3. Explore more XBraid settings in examples/ex-01-expanded.c

4. Explore advanced XBraid features
   → Adaptivity, residual and storage options, shell-vectors and BDF-k, spatial coarsening
   → Fortran90 Interface

5. Porting a user-code to XBraid with examples/ex-02
   → Debugging the connection to XBraid
   → Intrusiveness versus efficiency

6. A few application area highlights
**How to convert a user-code**

- **File**: examples/ex-02*
  - **Solves**: \( u_t = -u_{xx} \)

---

### ex-02-serial.c

```c
/* Set up simulation */
t= 0.0; tstop= 2*PI; ...

/* Initialize u(t=0) */
get_solution(values, ...);

/* Loop over all time values */
for(step=1; step < ntime; step++){
    t = t + deltaT;
    take_step(values, t, ...);
}

/* Process result */
compute_error_norm(values, ...);
save_solution(fname, values, ...);
```

### ex-02-lib.c

```c
/* Common functions with XBraid */

/* Initialization routine */
void get_solution(...);

/* Helpers for take_step */
void solve_tridiag(...);
void matvec_tridiag(...);
void compute_stencil(...);

/* Core time-stepping routine */
void take_step(...);

/* Output Functions */
double compute_error_norm(...);
void save_solution(...);

/* XBraid specific spatial interpolation/coarsening */
void interpolate_1D(...);
void coarsen_1D(...);
```

### ex-02.c

```bash
$ex-02-serial -ntime 64 -nspace 17
```
How to convert a user-code

- File: examples/ex-02*

Solves: $u_t = -u_{xx}$

**ex-02-serial.c**

```c
/* Set up simulation */
t= 0.0; tstop= 2*PI; ...

/* Initialize u(t=0) */
get_solution(values, ...);

/* Loop over all time values */
for(step=1; step < ntime; step++){
    t = t + deltaT;
    take_step(values, t, ...);
}

/* Process result */
compute_error_norm(values, ...);
save_solution(fname, values, ...);
```

**ex-02-lib.c**

```c
/* Common functions with XBraid */

/* Initialization routine */
void get_solution(...)

/* Helpers for take_step */
void solve_tridiag(...)
void matvec_tridiag(...)
void compute_stencil(...)

/* Core time-stepping routine */
void take_step(...)

/* Output Functions */
double compute_error_norm(...)
void save_solution(...)

/* XBraid specific spatial interpolation/coarsening */
void interpolate_1D(...)
void coarsen_1D(...)
```

**ex-02.c**

```
$ex-02-serial -ntime 64 -nspace 17
```

---

**ex-02-driver.c**

```
XBraid Driver...
```
# How to convert a user-code

- **File:** `examples/ex-02*

---

### ex-02-serial.c

```c
/* Set up simulation */
t = 0.0; tstop = 2*PI; ...

/* Initialize u(t=0) */
get_solution(values, ...);

/* Loop over all time values */
for(step=1; step < ntime; step++){
    t = t + deltaT;
    take_step(values, t, ...);
}

/* Process result */
compute_error_norm(values, ...);
save_solution(fname, values, ...);
```

### ex-02-lib.c

```c
/* Common functions with XBraid */

/* Initialization routine */
void get_solution(...)

/* Helpers for take_step */
void solve_tridiag(...)
void matvec_tridiag(...)
void computeStencil(...)

/* Core time-stepping routine */
void take_step(...)

/* Output Functions */
double compute_error_norm(...)
save_solution(...)

/* XBraid specific spatial interpolation/coarsening */
void interpolate_1D(...)
void coarsen_1D(...)
```

### ex-02.c

```c
XBraid Driver ...
```

---

$ex-02-serial -ntime 64 -nspace 17$

---

Solves: $u_t = -u_{xx}$
**How to convert a user-code**

- **File:** examples/ex-02*

Solves: \( u_t = -u_{xx} \)

---

**ex-02-lib.c**

- /* Common functions with XBraid */
- /* Initialization routine */
  
void get_solution(...)

- /* Helpers for take_step */
  
void solve_tridiag(...)
void matvec_tridiag(...)
void compute_stencil(...)

- /* Core time-stepping routine */
  
void take_step(...)

- /* Output Functions */
  
double compute_error_norm(...)
void save_solution(...)

- /* XBraid specific spatial interpolation/coarsening */
  
void interpolate_1D(...)
void coarsen_1D(...)

---

**ex-02.c**

```c
typedef struct _braid_App_struct
    MPI_Comm  comm;
    double    matrix[3];
    ...

typedef struct _braid_Vector_struct
    int     size;
    double *values;

int my_Step(u, ...)
take_step(u->values, ...);

int my_Access(u, ...)
compute_error_norm(u->values, ...);
save_solution(fname, u->values, ...);

int my_Init(u, ...)
get_solution(u->values, ...);`n
```

---

**App structure holds time-independent data for stepping**

---

**ex-serial.c**

Serial Driver...

---
How to convert a user-code

- **File:** examples/ex-02*

  **Solves:** \( u_t = -u_{xx} \)

```c
typedef struct _braid_App_struct
    MPI_Comm  comm;
    double    matrix[3];
...
```

```c
typedef struct _braid_Vector_struct
    int     size;
    double *values;
```

```c
int my_Step(u, ...)
take_step(u->values, ...);
```

```c
int my_Access(u, ...)
compute_error_norm(u->values, ...);
```

```c
int my_Init(u, ...)
get_solution(u->values, ...);
```

```c
main()
braid_Core core; app = (my_App *) ...
braid_Init(..., core);
braid_Drive(core);
```

Vector holds time-dependent data for stepping
**How to convert a user-code**

- **File:** examples/ex-02*

  Solves: $u_t = -u_{xx}$

<table>
<thead>
<tr>
<th><strong>ex-02-lib.c</strong></th>
<th><strong>ex-02.c</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>/* Common functions with XBraid */</td>
<td>typedef struct _braid_App_struct</td>
</tr>
<tr>
<td>/* Initialization routine */</td>
<td>MPI_Comm comm;</td>
</tr>
<tr>
<td>void get_solution(...)</td>
<td>double matrix[3];</td>
</tr>
<tr>
<td>/* Helpers for take_step */</td>
<td>...</td>
</tr>
<tr>
<td>void solve_tridiag(...)</td>
<td>typedef struct _braid_Vector_struct</td>
</tr>
<tr>
<td>void matvec_tridiag(...)</td>
<td>int size;</td>
</tr>
<tr>
<td>void compute_stencil(...)</td>
<td>double *values;</td>
</tr>
<tr>
<td>/* Core time-stepping routine */</td>
<td>int my_Step(u, ...)</td>
</tr>
<tr>
<td>void take_step(...)</td>
<td>take_step(u-&gt;values, ...);</td>
</tr>
<tr>
<td>/* Output Functions */</td>
<td>int my_Access(u, ...)</td>
</tr>
<tr>
<td>void compute_error_norm(...)</td>
<td>compute_error_norm(u-&gt;values, ...);</td>
</tr>
<tr>
<td>void save_solution(...)</td>
<td>save_solution(fname, u-&gt;values, ...);</td>
</tr>
<tr>
<td>/* XBraid specific spatial interpolation/coarsening */</td>
<td>int my_Init(u, ...)</td>
</tr>
<tr>
<td>void interpolate_1D(...)</td>
<td>get_solution(u-&gt;values, ...);</td>
</tr>
<tr>
<td>void coarsen_1D(...)</td>
<td>main()</td>
</tr>
<tr>
<td></td>
<td>braid_Core core; app = (my_App *) ...</td>
</tr>
<tr>
<td></td>
<td>braid_Init(..., core);</td>
</tr>
<tr>
<td></td>
<td>braid_Drive(core);</td>
</tr>
</tbody>
</table>

**ex-serial.c**

Serial Driver...

Various wrapper functions re-use library routines
How to convert a user-code

- **File:** examples/ex-02*

**Solves:** $u_t = -u_{xx}$

```c
typedef struct _braid_App_struct
    MPI_Comm  comm;
    double    matrix[3];
...

typedef struct _braid_Vector_struct
    int     size;
    double *values;

int my_Step(u, ...)
take_step(u->values, ...);

int my_Access(u, ...)
compute_error_norm(u->values, ...);
save_solution(fname, u->values, ...);

int my_Init(u, ...)
get_solution(u->values, ...);

main()
braid_Core core; app = (my_App *) ...
braid_Init(..., core);
braid_Drive(core);
```

Actually running XBraid is easy!
How to convert a user-code

- File: examples/ex-02*

Solves: $u_t = -u_{xx}$

```c
typedef struct _braid_App_struct
    MPI_Comm  comm;
    double    matrix[3];
...

typedef struct _braid_Vector_struct
    int     size;
    double *values;

int my_Step(u, ...)
take_step(u->values, ...);

int my_Access(u, ...)
compute_error_norm(u->values, ...);
save_solution(fname, u->values, ...);

int my_Init(u, ...)
get_solution(u->values, ...);

main()
braid_Core core; app = (my_App *) ...
braid_Init(..., core);
braid_Drive(core);
```

```
typedef struct _braid_App_struct
    MPI_Comm  comm;
    double    matrix[3];
...

typedef struct _braid_Vector_struct
    int     size;
    double *values;

int my_Step(u, ...)
take_step(u->values, ...);

int my_Access(u, ...)
compute_error_norm(u->values, ...);
save_solution(fname, u->values, ...);

int my_Init(u, ...)
get_solution(u->values, ...);

main()
braid_Core core; app = (my_App *) ...
braid_Init(..., core);
braid_Drive(core);
```

$ ex-02 -ntime 64 -nspace 17; python viz-ex-02.py
How to debug your new code

- **File:** examples/ex-02.c  
  Solves: $u_t = -u_{xx}$

<table>
<thead>
<tr>
<th>There is a test function for each wrapper, e.g., braid_TestInit()</th>
</tr>
</thead>
<tbody>
<tr>
<td>$./ex-02 -wrapper_tests</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Finished braid_TestAll: no fails detected</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set max_levels=1. The answer should exactly match sequential time stepping.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$./ex-02 -ntime 64 -nspace 17 -ml 1</td>
</tr>
<tr>
<td>$ python viz-ex-02.py</td>
</tr>
<tr>
<td>(In reality, you’d want to check the agreement to 15 or 16 decimals)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Continue with max_levels=1, but switch to multiple processors in time. Check that the answer again exactly matches sequential time stepping.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ mpirun -np 2 ex-02 -ntime 64 -nspace 17 -ml 1</td>
</tr>
<tr>
<td>$ python viz-ex-02.py</td>
</tr>
<tr>
<td>(In reality, you’d want to check the agreement to 15 or 16 decimals)</td>
</tr>
</tbody>
</table>
How to debug your new code

- **File:** `examples/ex-02.c`  
  **Solves:** $u_t = -u_{xx}$

**Check that XBraid is a fixed point method**

Set `max-levels=2`, `tol=0.0`, `max-iter=3`, and initialize XBraid with sequential soln.

```
$ ./ex-02 -ntime 64 -nspace 17 -ml 2 -tol 0.0 -mi 3 -use_seq
Braid: || r_0 || = 0.000000e+00, conv factor = 1.00e+00, wall time = ...
Braid: || r_1 || = 0.000000e+00, conv factor = nan, wall time = ...
Braid: || r_2 || = 0.000000e+00, conv factor = nan, wall time = ...
Braid: || r_3 || = 0.000000e+00, conv factor = nan, wall time = ...
Braid: || r_4 || = 0.000000e+00, conv factor = nan, wall time = ...
```
How to debug your new code

- File: examples/ex-02.c

Solves: $u_t = -u_{xx}$

Turn on debug-level printing and check that the exact solution is propagating.
With FCF-relaxation, the exact solution propagates forward 2 C-points each iter.

```
$.ex-02 -ntime 8 -nspace 17 -mi 3 -print_level 2
Braid: time step: 0, rnorm: 0.00e+00
Braid: time step: 2, rnorm: 0.00e+00
Braid: time step: 4, rnorm: 6.86e-01
Braid: time step: 6, rnorm: 1.10e+00
Braid: time step: 8, rnorm: 2.04e-02
Braid: || r_0 || = 1.292837e+00, conv factor = 1.00e+00, wall time = ...
Braid: time step: 0, rnorm: 0.00e+00
Braid: time step: 2, rnorm: 0.00e+00
Braid: time step: 4, rnorm: 0.00e+00
Braid: time step: 6, rnorm: 0.00e+00
Braid: time step: 8, rnorm: 1.62e-02
...
```

Then, run some larger, multilevel tests of XBraid, checking that the sequential and time-parallel versions agree to within the halting tolerance.
Intrusiveness versus efficiency

- The more intrusive XBraid is allowed to be, the more efficient it is.
  - **Residual option:** computing the residual with a naive implementation of XBraid is as expensive in FLOPs as sequential time stepping. Writing this extra function allows you to avoid this for implicit schemes.
    - This function also allows relaxation to be significantly less expensive
  - **Adaptivity:** constructing the correct adaptive space-time grid is active research.
    - For instance a development branch is currently using threshold refinement across the temporal communicator to choose time intervals to refine.
  - **Storage:** requires a little extra coding, i.e., a new initial guess to implicit scheme
  - **Level-dependent time-stepper:** how to change `Step()` on coarse-levels is problem dependent, but almost always yields big benefits, e.g., vary the tolerance
  - **Spatial coarsening:** this can affect convergence, but is required for an $O(N)$ method in time and space.
  - **Stephanie Friedhoff’s talk** covers this in more detail, e.g., results from taking a naive XBraid implementation and moving to an STMG (space-time MG) method.
Outline

1. Introduction
   → Tutorial software requirements and XBraid overview

2. Simplest example of solving a scalar ODE with examples/ex-01
   → Defining the App and vector structures, writing wrapper functions, running XBraid

3. Explore more XBraid settings in examples/ex-01-expanded.c

4. Explore advanced XBraid features
   → Adaptivity, residual and storage options, shell-vectors and BDF-k, spatial coarsening
   → Fortran90 Interface

5. Porting a user-code to XBraid with examples/ex-02
   → Debugging the connection to XBraid
   → Intrusiveness versus efficiency

6. A few application area highlights
Experiments coupling our code XBraid with various application research codes

- Navier-Stokes (compressible and incompressible)
  - Strand2D, CarT3D, LifeV (Trilinos-based)

- Heat equation (including moving mesh example)
  - MFEM, hypre

- Nonlinear diffusion, the $p$-Laplacian
  - MFEM

- Power-grid simulations (project just starting)
  - GridDyn

- Explicit time-stepping coupled with space-time coarsening
  - Heat equation
  - Advection plus artificial dissipation
  - MFEM, hypre
Compressible Navier-Stokes (nonlinear) – speedups to 7.5x with typical MG scaling

- Coupled XBraid with existing code Strand2D (DoD project)
  - ~500 lines of XBraid wrapper code plus minor changes to Strand2D
  - ~3 weeks with minimal outside help

- Plots of velocity magnitude at time step 5120
Compressible Navier-Stokes with Cart3D – convergence is very fast, ~5 iterations

- Taylor-Green problem: turbulent decay of vortex, $Re=1600$
  - Higher-order spatial discretization on $58^3 \times 20,000$ cartesian grid
- Plot velocity magnitude at $x=0$ cross-section
Strong scaling for heat equation

- XBraid uses V-cycles and FCF-relaxation
- Excellent strong scaling, until parallelism is exhausted

![Graph showing strong scaling for heat equation](image-url)

- 257^2 \times 16384 space-time grid
- Max speedup is 52
- Cross-over at \approx 32 cores
The $p$-Laplacian: nonlinear diffusion

- Solve $u_t = \nabla \cdot (|\nabla u|^{p-2} \nabla u)$
- 2D linear finite elements
  - 16K x 20K space-time problem
  - Backward Euler (Newton’s method)
- Current results
  - Crossover at ~40 processors in time
  - Speedup of 18x at 130K cores
- Important parameters for performance
  - Full storage and space-time coarsening
  - Adjusting the Newton tolerance for the early iterations

Initial speedups for power-grid

- Simulate 4 generators (30 unknowns) for 30s with 30K time steps

<table>
<thead>
<tr>
<th></th>
<th>Sequential</th>
<th>128 cores</th>
<th>256 cores</th>
<th>512 cores</th>
<th>1024 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit RK4</td>
<td>227s</td>
<td>144s</td>
<td>113s</td>
<td><strong>102s</strong></td>
<td>105s</td>
</tr>
<tr>
<td>BDF-4</td>
<td>12.4s</td>
<td>13.6s</td>
<td>9.46s</td>
<td>7.73s</td>
<td><strong>7.30s</strong></td>
</tr>
</tbody>
</table>

- XBraid is designed for one-step methods, so we make BDF-k “one-step” by grouping k time-steps together
  - Creates non-uniform time-step sizes on coarse grids and stability issues for $\Phi$

Solution: reduce the BDF order on coarse levels
Explicit methods with MFEM

- 2D advection $u_t = b(x) \cdot \nabla u + \gamma \Delta u$
  - Stability determined by convection (convection dominated)
  - Diffusion term 0.001

- **Sequential Time Stepping**
  - Sharp profile is transported over 1100 time steps
  - 3rd order explicit method
Explicit methods with MFEM

- 2D advection $u_t = b(x) \cdot \nabla u + \gamma \Delta u$
  - Stability determined by convection (convection dominated)
  - Diffusion term 0.001

- **Parallel-in-time solution**
  - Sharp profile is transported over 1100 time steps
  - 3\textsuperscript{rd} order explicit method
  - 3-level XBraid hierarchy
Explicit methods with MFEM

- 2D advection \( u_t = b(x) \cdot \nabla u + \gamma \Delta u \)
  - Stability determined by convection (convection dominated)
  - Diffusion term 0.001

- Parallel-in-time solution
  - Sharp profile is transported over 1100 time steps
  - 3\textsuperscript{rd} order explicit method
  - 3-level XBraid hierarchy
Explicit methods with MFEM

- 2D advection $u_t = b(x) \cdot \nabla u + \gamma \Delta u$
  - Stability determined by convection (convection dominated)
  - Diffusion term 0.001
- Parallel-in-time solution
  - Sharp profile is transported over 1100 time steps
  - 3rd order explicit method
  - 3-level XBraid hierarchy
Explicit methods with MFEM

- 2D advection $u_t = b(x) \cdot \nabla u + \gamma \Delta u$
  - Stability determined by convection (convection dominated)
  - Diffusion term 0.001
- Parallel-in-time solution
  - Sharp profile is transported over 1100 time steps
  - 3rd order explicit method
  - 3-level XBraid hierarchy
Explicit methods with MFEM

- **2D advection** \( u_t = b(x) \cdot \nabla u + \gamma \Delta u \)
  - Stability determined by convection (convection dominated)
  - Diffusion term 0.001
- **Parallel-in-time solution**
  - Sharp profile is transported over 1100 time steps
  - 3rd order explicit method
  - 3-level XBraid hierarchy

\[ u_t = b(x) \cdot \nabla u + \gamma \Delta u \]
Explicit methods with MFEM

- **2D advection** \( u_t = b(x) \cdot \nabla u + \gamma \Delta u \)
  - Stability determined by convection (convection dominated)
  - Diffusion term 0.001

- **Parallel-in-time solution**
  - Sharp profile is transported over 1100 time steps
  - 3rd order explicit method
  - 3-level XBraid hierarchy

- **Future Work:** Improve convergence (relaxation, coarse-grid equations)
Moving mesh

- 1D space moving mesh proof-of-concept (nonlinear parabolic prob.)
- Mesh points move towards regions with a rapidly changing solution
  - Heat sources force mesh to move
- Fast convergence and scalable iteration counts
- More complicated moving mesh problems coming...
Temporal adaptivity proof-of-concept

- Classic ODE modeling satellite orbit around earth and moon (2 variables in space, $x$ and $y$)
- One region of orbit requires very fine time steps
  - Carry out 4 periods of orbit, refining step size as needed
Nearly 50 years of research exists, but has only scratched the surface

- **Earliest work** goes back to 1964 by Nievergelt
  - Led to multiple shooting methods, Keller (1968)

- **Space-time multigrid** methods for parabolic problems
  - The latter is one of the first **optimal & fully parallelizable** methods to date

- **Parareal** was introduced by Lions, Maday, and Turincini in 2001
  - Probably the most widely studied method
  - Gander and Vandewalle (2007) show that parareal is **two-level FAS multigrid**

- **Discretization specific** work includes
  - DeSterck, Manteuffel, McCormick, Olson (2004, 2006) – FOSLS

- **Research on these methods is ramping up!**
  - Ruprecht, Krause, Speck, Emmett, Langer, … **this is not an exhaustive list**
Summary and conclusions

- Sequential time integration bottleneck is real
  - Parallel in time is needed for future architectures
  - This is a major paradigm shift

- XBraid applies multigrid reduction to the time dimension
  - Multigrid is ideal for exascale (optimal, resilient, ...)
  - Result is a flexible and non-intrusive approach

- The more intrusive XBraid is allowed to be, the more efficient the algorithm is.

- There is much future work to be done!
  - More problem types, more complicated discretizations, performance improvements, adaptive meshing, ...
Thank You! Any Questions?


Open Source XBraid Code
- [http://llnl.gov/casc/xbraid](http://llnl.gov/casc/xbraid)
- Supports C, C++, F90

Our Team

Collaborators and summer interns
CU Boulder (Manteuffel, McCormick, O’Neill, Southworth), Memorial University (MacLachlan), U Cologne (Friedhoff), U Stuttgart (Hessenthaler), Monash U (De Sterck)