XBraid Tutorial

A flexible and scalable approach to parallel-in-time

Jacob Schroder and Rob Falgout

NSF CBMS PinT Summer School
Michigan Tech, Houghton, Michigan
August, 2022

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. Porting a user-code to XBraid with `examples/ex-02`
   → Debugging the connection to XBraid
   → Intrusiveness versus efficiency
5. A few application area highlights

Appendix: *Advanced XBraid features*

- *Temporal adaptivity*
- *Shell-vectors and BDF-k*
- *Fortran90 Interface*
- *Residual and storage options*
- *Spatial coarsening*
- *Python Interface*

To interact with the tutorial, you need

- A working installation of XBraid
  `https://github.com/XBraid/xbraid`
  - Github home page has basic information on installation
  - User's manual has more comprehensive information
    https://github.com/XBraid/xbraid/files/5144094/user_manual.pdf

- XBraid *required (repository head)*
- GCC compiler *required*
- MPI *recommended*
- Python 3 with NumPy, Matplotlib *recommended*
- *hypre* installation for running example `ex-03` *optional*
  `https://github.com/hypre-space/hypre`

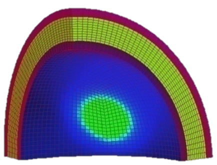
To interact with the tutorial, you need

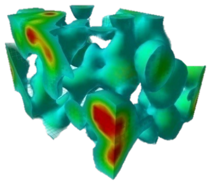
- Make sure you can run

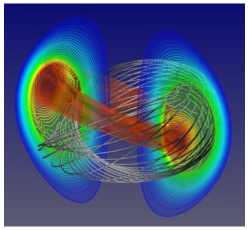
```
$ cd xbraid
$ make
$ cd examples
$ make ex-01 ex-02
$ ./ex-01
 Braid: || r_1 || = 2.845538e-02, conv factor = 1.00e+00, wall time = ...
 Braid: || r_2 || = 8.621939e-04, conv factor = 3.03e-02, wall time = ...
 Braid: || r_3 || = 0.000000e+00, conv factor = 0.00e+00, wall time = ...
 ...
$ ./ex-02
 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 = ...
 ...
```

Multigrid is well suited for exascale

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


Elasticity / Plasticity


Quantum Chromodynamics

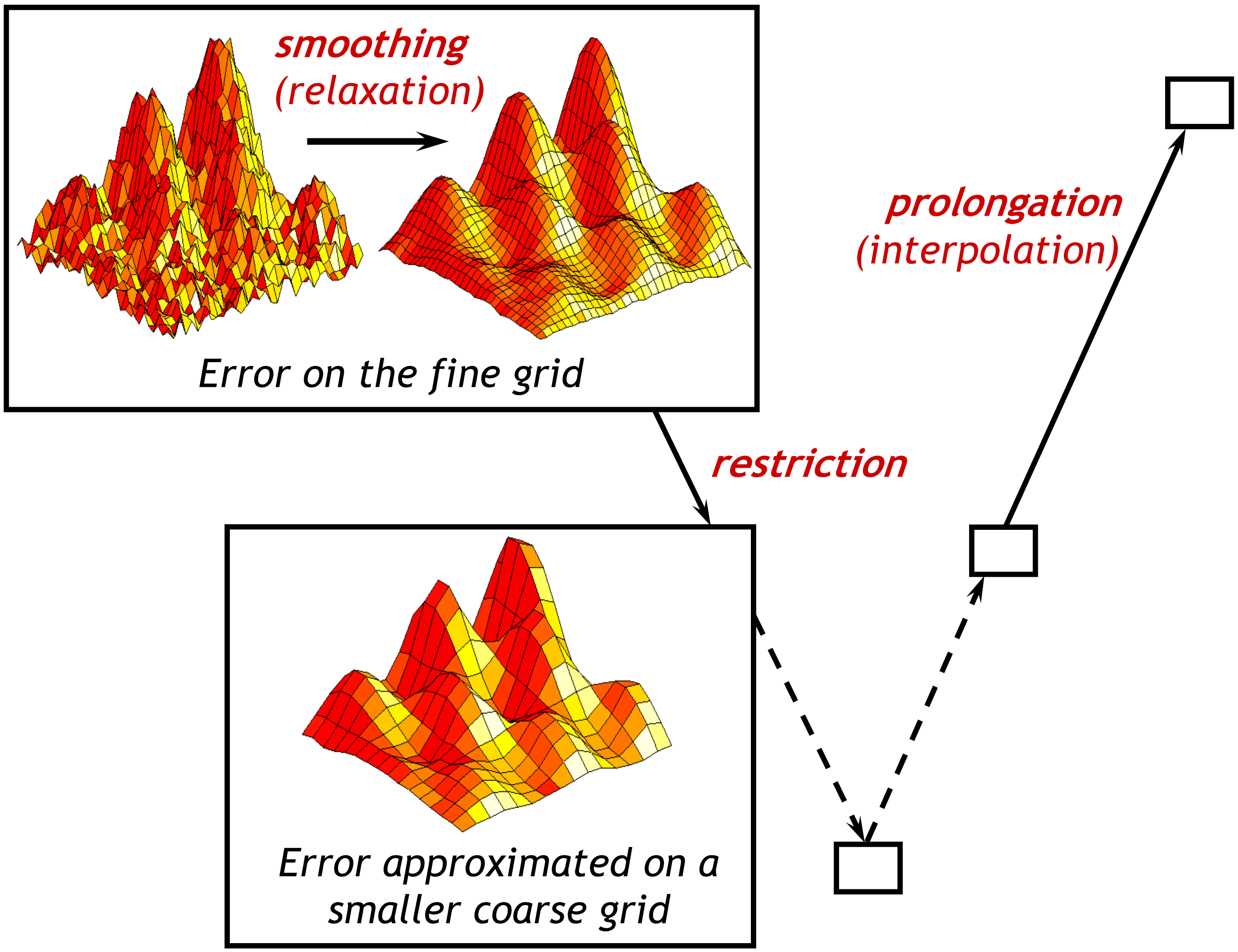

Magnetic Fusion Energy

- Exascale solver algorithms will need to:
  - Exhibit extreme levels of parallelism (exascale → 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 heterogeneity, then so can multigrid
  - Be resilient to faults
    Multigrid has already shown good resilience (iterative and multilevel helps)

Parallel-in-time approach: Leverage spatial multigrid research

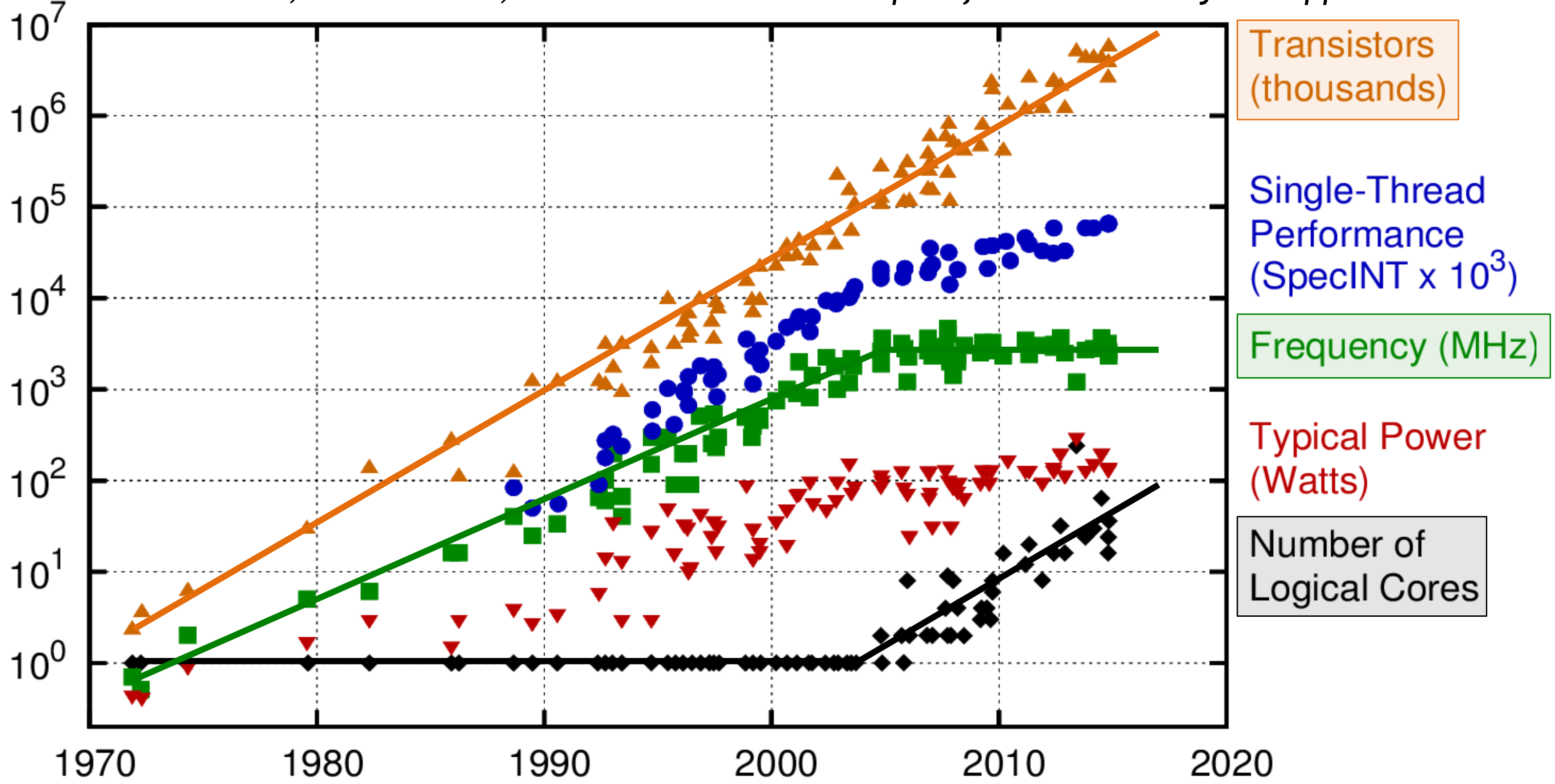
Solve:

$A(u) = b$



Parallel time integration: Paradigm shift driven by computer architecture trends

Data from 1970-2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data and plot for 2010-2015 by K. Rupp.



- Architecture trend: clock rates are no longer increasing – faster speed is now achieved through more concurrency
- Parallel time integration methods are needed (think exascale)!

Technical approach

- Consider the **general** one-step method

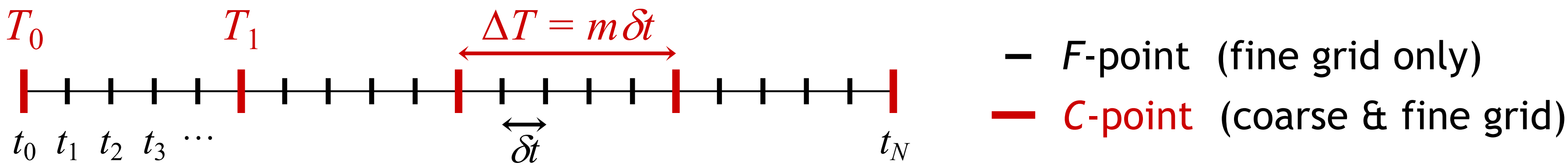
$$\boldsymbol{u}_i = \Phi_i(\boldsymbol{u}_{i-1}) + \boldsymbol{g}_i, \quad i = 1, 2, ..., N$$

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

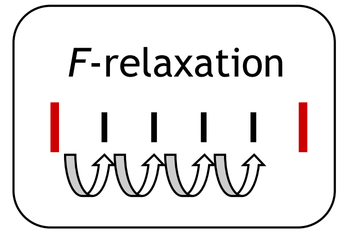
$$A\mathbf{u} \equiv \begin{pmatrix} I & & & \\ -\Phi & I & & \\ & \ddots & \ddots & \\ & & -\Phi & I \end{pmatrix} \begin{pmatrix} \boldsymbol{u}_0 \\ \boldsymbol{u}_1 \\ \vdots \\ \boldsymbol{u}_N \end{pmatrix} = \begin{pmatrix} \boldsymbol{g}_0 \\ \boldsymbol{g}_1 \\ \vdots \\ \boldsymbol{g}_N \end{pmatrix} \equiv \mathbf{g}$$

- Instead solve 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

Multigrid reduction in time (MGRIT)[1]



- Relaxation is highly parallel
  - Block-Jacobi alternating between *F*-points and *C*-points



- Coarse system is a time rediscretization with N/m block rows
  - Approximate impractical Φ^m with Φ_Δ *(some rediscretization with ΔT)*

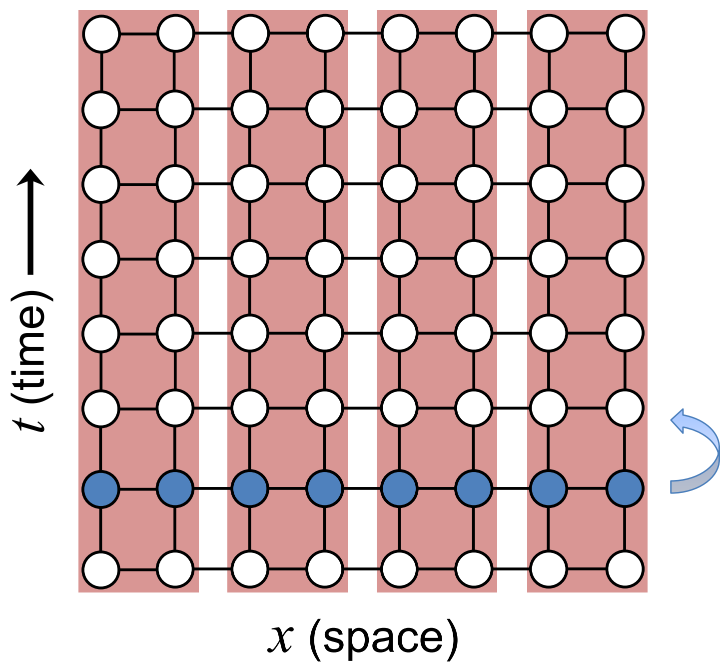
$$A_\Delta = \begin{pmatrix} I & & & \\ -\Phi^m & I & & \\ & \ddots & \ddots & \\ & & -\Phi^m & I \end{pmatrix} \quad \Rightarrow \quad B_\Delta = \begin{pmatrix} I & & & \\ -\Phi_\Delta & I & & \\ & \ddots & \ddots & \\ & & -\Phi_\Delta & I \end{pmatrix}$$

  - Apply recursively for multilevel hierarchy

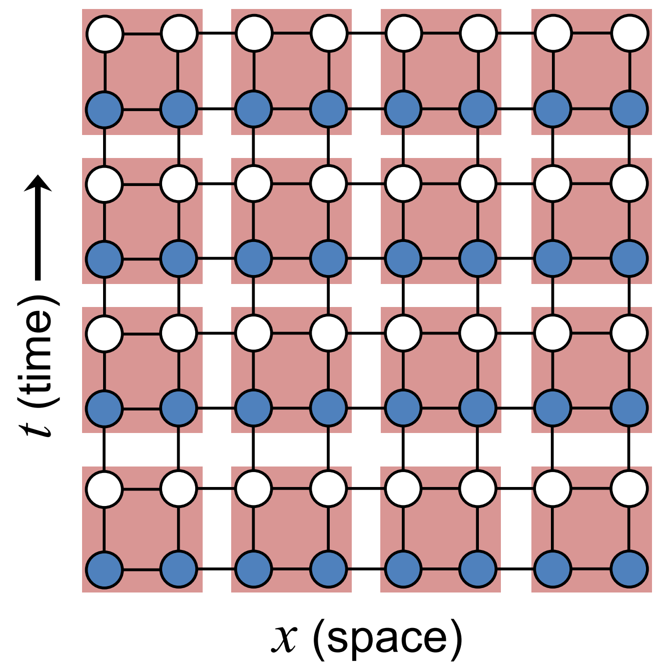
1. Falgout, Friedhoff, Kolev, MacLachlan, Schroder, *Parallel Time Integration with Multigrid*, SISC, 2014.

Parallel decomposition

Serial time stepping



Multigrid in time



Minus: Parallelize in space only
Plus: Store only one time step

Plus: Parallelize in space and time
Minus: Store several time steps, but per processor costs still similar

Pink regions denote one processor

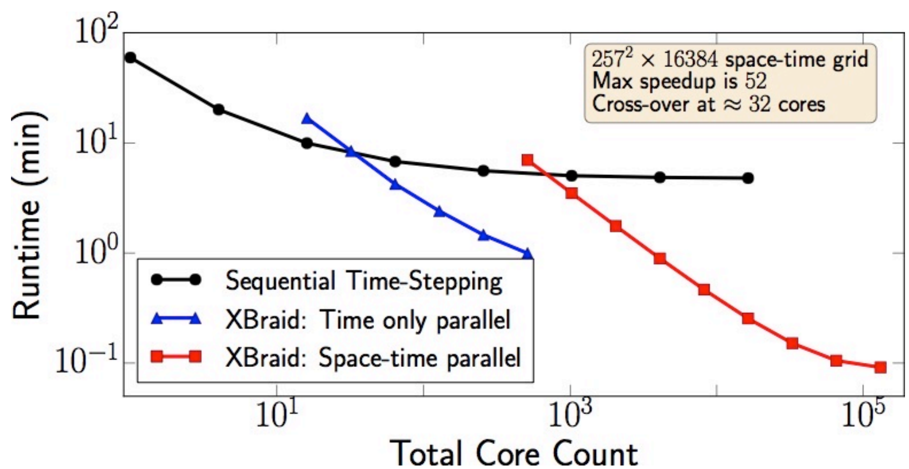
A broad summary of MGRIT

- Expose **concurrency** in the time dimension with multigrid
- **Non-intrusive**, with unchanged fine-grid problem
- Converges to **same solution** as sequential marching

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

- Optimal for variety of parabolic problems
- Extends to **nonlinear** problems with FAS formulation
- In simple two-level setting, MGRIT $\equiv$ Parareal

- Large speedups available, but in a new way
  - Time stepping is already O(N)
  - Useful only beyond a crossover
  - More time steps → more speedup potential
  - *XBraid* is our MGRIT code



1. Falgout, Friedhoff, Kolev, MacLachlan, Schroder, *Parallel Time Integration with Multigrid*, SISC, 2014.

XBraid: Open source, non-intrusive, and flexible

- User writes several wrapper routines:
  - *`Step`*, *`Init`*, *`Clone`*, *`Sum`*, *`SpatialNorm`*, *`Access`*, *`BufPack`*, *`BufUnpack`*
  - *`Coarsen`*, *`Refine`* (optional, for spatial coarsening)

- Example: *`Step`(app, **u**, status)*
  - Advance vector ***u*** from time *tstart* to *tstop*

- Code stores only *C*-points to minimize storage
  - Memory multiplier per processor:
    ~*O(log N)* with time coarsening, *O(1)* with space-time coarsening

- Processes time-intervals to overlap communication and computation
- Supports adaptivity in time and space

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

5. A few application area highlights

Appendix: *Advanced XBraid features*

- *Temporal adaptivity*
- *Shell-vectors and BDF-k*
- *Fortran90 Interface*
- *Residual and storage options*
- *Spatial coarsening*
- *Python Interface*

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).

```
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 one double.

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


typedef struct _braid_Vector_struct{
   double value;
} my_Vector;
```

Define the `Step()` function

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

`Step()` evolves `u` from `tstart` to `tstop`

```
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: $u_t = \lambda u$

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: $u_t = \lambda u$

Vector at `tstop` from previous XBraid iteration (initial guess for implicit solvers)

```
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: $u_t = \lambda u$

Vector at `tstart`

```
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: $u_t = \lambda u$

Ignore by default. (XBraid forcing term, only needed if residual option is used)

```
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: $u_t = \lambda u$

Status structures can be queried for various information (level, iteration, etc...)

```
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: $u_t = \lambda u$

For instance, to get `tstart, tstop`

```
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

```
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

```
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 simple 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;
}
```

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**

```
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

```
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`

`BufPack()` flattens the vector `u` into `buffer`

```
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`

Packing this buffer entails just setting a single double value

```
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 (the buffer size) with a status structure

```
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()`

```
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()`

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

```
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

```
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()`

Initialize App structure

```
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()`

Initialize `braid_Core`, passing in all user-written functions

```
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 XBraid in `main()`

Set all the XBraid options that you want

```
int main()
  ...
  braid_SetPrintLevel( core, 2);
  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 XBraid in `main()`

Run the simulation

```
int main()
   ...
   braid_SetPrintLevel( core, 2);
   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 XBraid in `main()`

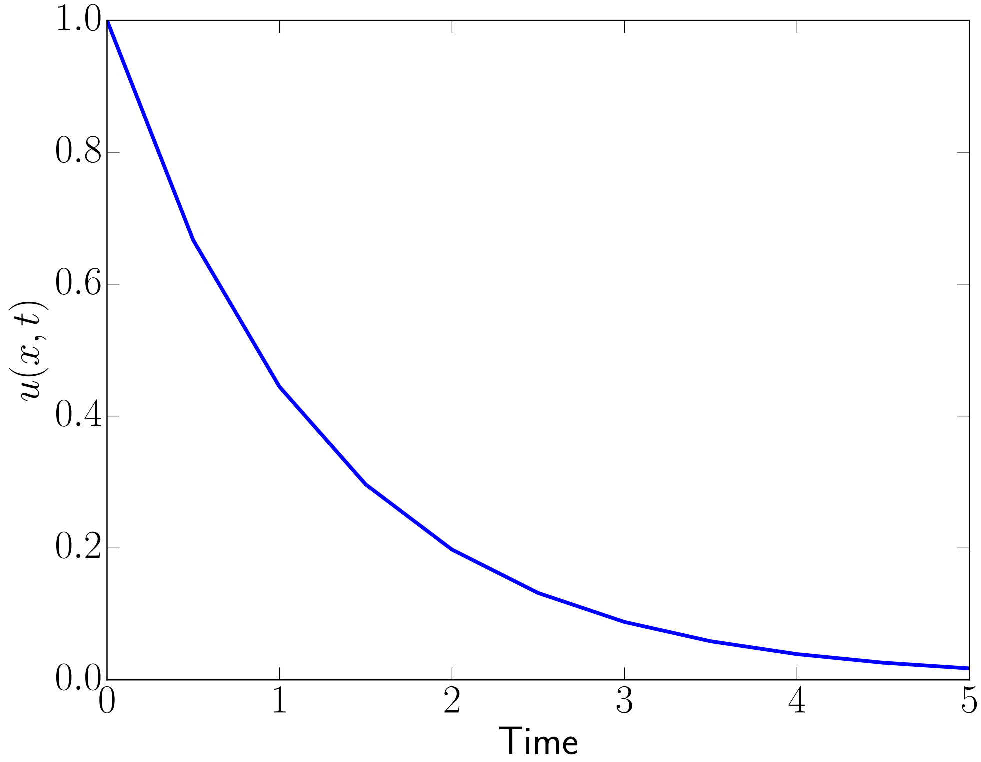
Clean up

```
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` Solves: $u_t = \lambda u$
- 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
```



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

5. A few application area highlights

Appendix: *Advanced XBraid features*

- *Temporal adaptivity*
- *Shell-vectors and BDF-k*
- *Fortran90 Interface*
- *Residual and storage options*
- *Spatial coarsening*
- *Python Interface*

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 some of these options!

```
$ 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
 ...
```

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 = ...

 Braid Solver Stats:
 start time = 0.000000e+00
 stop time  = 5.000000e+00
 time steps = 10

 use seq soln?          = 0
 storage                = -1

 max iterations         = 100
 iterations             = 4

 residual norm          = 0.000000e+00
 stopping tolerance     = 1.000000e-06
 use relative tol?      = 0
```

Examine the standard XBraid output

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

Basic time domain information

```
$ ./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 = ...

 Braid Solver Stats:
 start time = 0.000000e+00
 stop time  = 5.000000e+00
 time steps = 10

 use seq soln?          = 0
 storage                = -1

 max iterations         = 100
 iterations             = 4

 residual norm          = 0.000000e+00
 stopping tolerance     = 1.000000e-06
 use relative tol?      = 0
```

Examine the standard XBraid output

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

Advanced options

```
$ ./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 = ...

 Braid Solver Stats:
 start time = 0.000000e+00
 stop time  = 5.000000e+00
 time steps = 10

 use seq soln?           = 0
 storage                 = -1

 max iterations          = 100
 iterations              = 4

 residual norm           = 0.000000e+00
 stopping tolerance      = 1.000000e-06
 use relative tol?       = 0
```

Examine the standard XBraid output

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

Max allowed XBraid iterations

```
$ ./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 = ...

 Braid Solver Stats:
 start time = 0.000000e+00
 stop time  = 5.000000e+00
 time steps = 10

 use seq soln?          = 0
 storage                = -1

 max iterations         = 100
 iterations             = 4

 residual norm          = 0.000000e+00
 stopping tolerance     = 1.000000e-06
 use relative tol?      = 0
```

Examine the standard XBraid output

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

XBraid iterations taken

```
$ ./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 = ...

 Braid Solver Stats:
 start time = 0.000000e+00
 stop time  = 5.000000e+00
 time steps = 10

 use seq soln?         = 0
 storage               = -1

 max iterations        = 100
 iterations            = 4

 residual norm         = 0.000000e+00
 stopping tolerance    = 1.000000e-06
 use relative tol?     = 0
```

Examine the standard XBraid output

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

XBraid final residual norm and halting tolerance

```
$ ./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 = ...

 Braid Solver Stats:
 start time = 0.000000e+00
 stop time  = 5.000000e+00
 time steps = 10

 use seq soln?          = 0
 storage                = -1

 max iterations         = 100
 iterations             = 4

 residual norm          = 0.000000e+00
 stopping tolerance     = 1.000000e-06
 use relative tol?      = 0
```

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
 periodic                 = 0
 relax_only_cg            = 0
 finalFCRelax             = 0
 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: $u_t = \lambda u$

Describe the XBraid options for setting number of levels / how far to coarsen

```
$ ./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
 periodic                 = 0
 relax_only_cg            = 0
 finalFCRelax             = 0
 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: $u_t = \lambda u$

Advanced XBraid options, e.g., periodic problem, num adaptive refinements, ...

```
$ ./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
 periodic                 = 0
 relax_only_cg            = 0
 finalFCRelax             = 0
 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

```
$ ./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
 periodic                   = 0
 relax_only_cg          = 0
 finalFCRelax           = 0
 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 || = 3.530338e-05, conv factor = 3.39e-02, wall time = ...
 Braid: || r_4 || = 3.716892e-07, conv factor = 1.05e-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 || = 1.049429e-03, conv factor = 3.68e-02, wall time = ...
 Braid: || r_3 || = 4.437913e-05, conv factor = 4.23e-02, wall time = ...
 Braid: || r_4 || = 1.990483e-06, conv factor = 4.49e-02, wall time = ...
 Braid: || r_5 || = 9.174722e-08, conv factor = 4.61e-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 = 1.01e-01, 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 || = 5.631827e-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
...
```

Don't over solve your problem

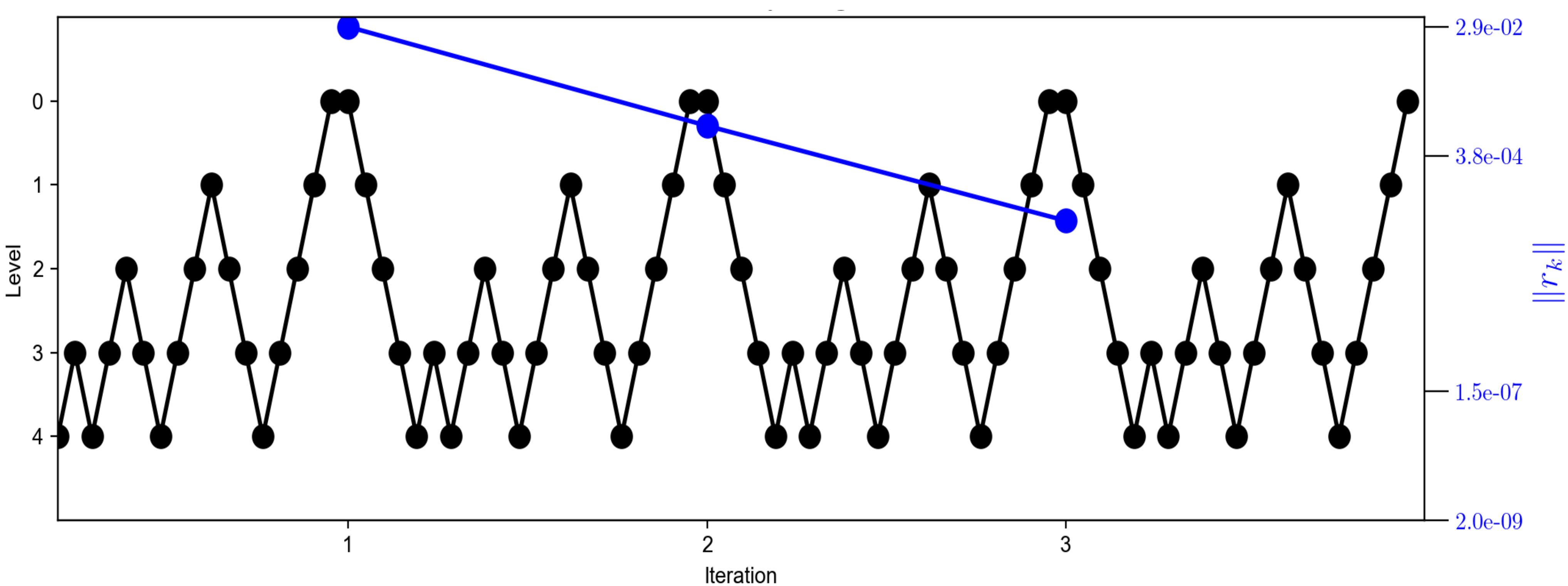
Full multigrid cycles (FMG)

- 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 ../misc/user_utils/cycleplot.py
```

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

5. A few application area highlights

Appendix: *Advanced XBraid features*

- *Temporal adaptivity*
- *Shell-vectors and BDF-k*
- *Fortran90 Interface*
- *Residual and storage options*
- *Spatial coarsening*
- *Python Interface*

How to convert a user-code

- File: `examples/ex-02*`

Solves: $u_t = u_{xx}$

ex-02-serial.c

```
/* Define space-time domain */
tstart= 0.0; tstop= 2*PI; ...

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


/* Main time step loop */
for(step=1; step <= ntime; step++){
   t = t + deltaT;
   take_step(values, t, ...);

   /* Output Solution */
   save_solution(filename, ...);
}

error = compute_error_norm(...);
```

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

ex-02-lib.c

Shared functions for serial and 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

```
XBraid
Driver
...
```

How to convert a user-code

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

ex-02-serial.c

```
/* Define space-time domain */
tstart= 0.0; tstop= 2*PI; ...

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


/* Main time step loop */
for(step=1; step <= ntime; step++){
   t = t + deltaT;
   take_step(values, t, ...);

   /* Output Solution */
   save_solution(filename, ...);
}

error = compute_error_norm(...);
```

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

ex-02-lib.c
Shared functions for serial and 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

```
XBraid
Driver
...
```

How to convert a user-code

- File: `examples/ex-02*`

Solves: $u_t = u_{xx}$

ex-02-serial.c

```
/* Define space-time domain */
tstart= 0.0; tstop= 2*PI; ...

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


/* Main time step loop */
for(step=1; step <= ntime; step++){
   t = t + deltaT;
   take_step(values, t, ...);

   /* Output Solution */
   save_solution(filename, ...);
}

error = compute_error_norm(...);
```

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

ex-02-lib.c
Shared functions for serial and 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

```
XBraid
Driver
...
```

How to convert a user-code

- File: `examples/ex-02*`

Solves: $u_t = u_{xx}$

ex-02-serial.c

```
/* Define space-time domain */
tstart= 0.0; tstop= 2*PI; ...

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


/* Main time step loop */
for(step=1; step <= ntime; step++){
   t = t + deltaT;
   take_step(values, t, ...);

   /* Output Solution */
   save_solution(filename, ...);
}

error = compute_error_norm(...);
```

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

ex-02-lib.c

Shared functions for serial and XBraid

```
/* Initialization array of values*/
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

```
XBraid
Driver
...
```

How to convert a user-code

- File: `examples/ex-02*`

Solves: $u_t = u_{xx}$

ex-serial.c

```
Serial
Driver
...
```

ex-02-lib.c
Shared functions for serial and 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

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

typedef struct _braid_Vector_struct
 int      size;
 double *values;  // vector at time t

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);
```

App structure holds time-independent data for stepping

How to convert a user-code

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

ex-serial.c

```
Serial
Driver
...
```

ex-02-lib.c
Shared functions for serial and 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

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

typedef struct _braid_Vector_struct
 int      size;
 double *values;  // vector at time t

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);
```

Vector holds time-dependent data for stepping

How to convert a user-code

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

ex-serial.c

```
Serial
Driver
...
```

ex-02-lib.c
Shared functions for serial and 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

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

typedef struct _braid_Vector_struct
 int      size;
 double *values;  // vector at time t

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);
```

Various wrapper functions re-use library routines

How to convert a user-code

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

ex-serial.c

```
Serial
Driver
...
```

ex-02-lib.c
Shared functions for serial and 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

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

typedef struct _braid_Vector_struct
 int       size;
 double *values;   // vector at time t

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: $u_t = u_{xx}$

ex-serial.c

```
Serial
Driver
...
```

ex-02-lib.c
Shared functions for serial and 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

```
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
```

Run code in parallel -- Speed up!

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

Run sequential baseline

```
$ ./ex-02 -nspace 1025 -ntime 1024 -ml 1
  → 0.45s
    Discretization error at final time:  1.9145e-03
```

Run Parareal

```
$ mpirun -np 6 ex-02 -nspace 1025 -ntime 1024 -ml 2 -tol 1e-4 -nu 0 -cf 16
  → 0.19s, 7 iterations
    Discretization error at final time:  1.9146e-03
```

Run MGRIT (still two-level, but with FCF)

```
$ mpirun -np 6 ex-02 -nspace 1025 -ntime 1024 -ml 2 -tol 1e-4 -nu 1 -cf 16
  → 0.19s, 4 iterations
    Discretization error at final time:  1.9125e-03
```

Run MGRIT with Richardson extrapolation in time (still two-level, but with FCF)

```
$ mpirun -np 6 ex-02 -nspace 1025 -ntime 1024 -ml 2 -tol 1e-4 -nu 0 -cf 16 -richardson
  → 0.20s, 4 iterations
    Discretization error at final time:  6.1440e-05
```

For larger problems, can go to more levels, further tune coarsening factor (cf), and so on...

How to debug your new code

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

Set `max-levels=1`. The answer should exactly match sequential time stepping.

```
$ ./ex-02 -ntime 64 -nspace 17 -ml 1
$ python viz-ex-02.py

 In practice, you want to check that the above XBraid run and a
 seperate sequential time-stepping run agree to 15 or 16 decimals
```

Continue with `max-levels=1`, but switch to multiple processors in time.
→ Check that the answer again exactly matches sequential time stepping.

```
$ mpirun -np 2 ex-02 -ntime 64 -nspace 17 -ml 1
$ python viz-ex-02.py
```

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 the sequential solution

```
$ ./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 3
 Braid:  time step:      0, rnorm: 0.00e+00
 Braid:  time step:      2, rnorm: 0.00e+00
 Braid:  time step:      4, rnorm: 5.61e-01
 Braid:  time step:      6, rnorm: 1.23e+00
 Braid:  time step:      8, rnorm: 1.86e-02
 Braid: || r_0 || = 1.355604e+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: 1.33e-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
    - Creates a method closer to Gander/Neumueller
    - Further modifications can result in a method similar to space-time MG
  - **Adaptivity:** adaptively refine in time and space, building new MGRIT levels
  - **Storage:** store all time-steps (C and F), provides improved initial guess for implicit scheme
  - **Level-dependent time-stepper:** Change `Step()` on coarse-levels for efficiency (problem dependent), e.g., vary implicit solve tolerance in `Step()`
  - **Spatial coarsening:** this can affect convergence, but is required for an $O(N)$ method in both time and space

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

5. A few application area highlights

Appendix: *Advanced XBraid features*

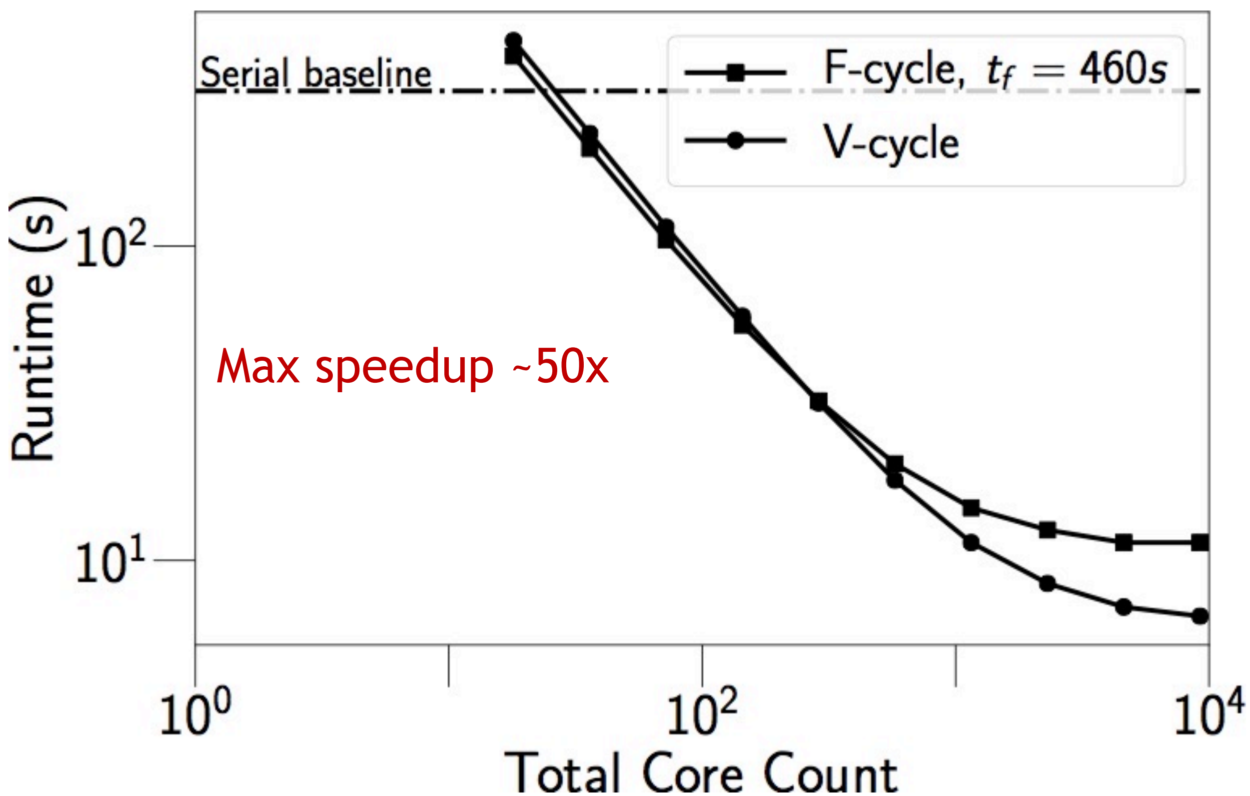
- *Temporal adaptivity*
- *Shell-vectors and BDF-k*
- *Fortran90 Interface*
- *Residual and storage options*
- *Spatial coarsening*
- *Python Interface*

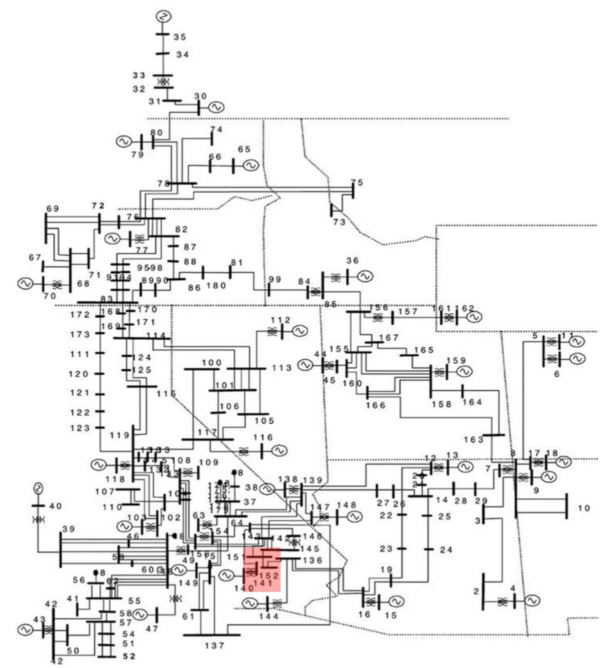
Experiments coupling our code XBraid with various application research codes

- Navier-Stokes (compressible and incompressible), Shallow Water
  - Strand2D, CarT3D, Cyclops, Chord
- Heat equation (including moving mesh example)
  - MFEM, hypre
- Elasticity (e.g., cardiac modeling)
  - CHeart
- Nonlinear diffusion, the *p*-Laplacian
  - MFEM
- Power-grid simulations
  - GridDyn+SunDials
- Explicit time-stepping coupled with space-time coarsening
  - Advection, Burger's Equation
  - MFEM
- Optimization (XBraid-adjoint), Machine Learning
  - CoDiPack, TorchBraid

Powergrid (DAE)

- Discontinuous square pulse applied to bus 141 every second[1]
  - Must handle discontinuities (events) for real-world relevance
  - Explore scalability w.r.t. number of discontinuities, 460s simulation has 460 events
  - Adaptively refine in time around discontinuities for improved accuracy





WECC System: 179 buses and 793 unknowns

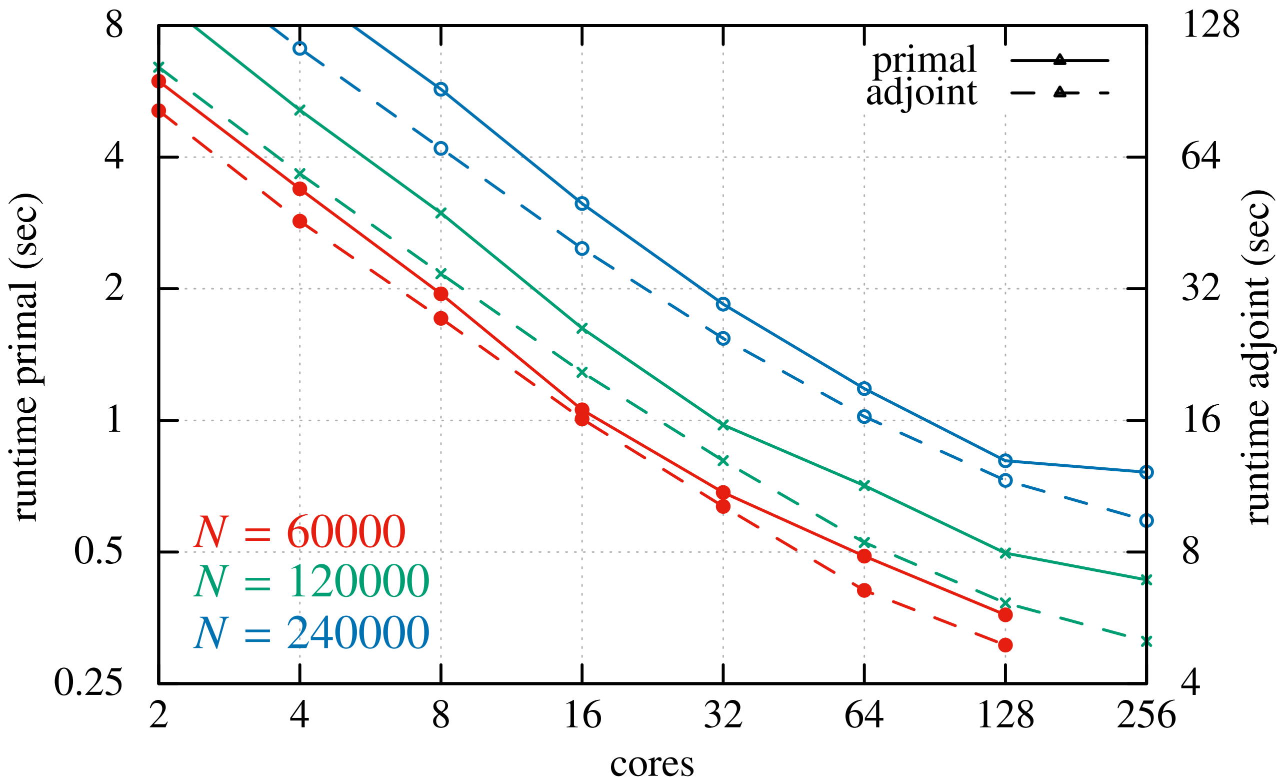
1. Schroder, Lecouvez, Falgout, Woodward, Top, *Parallel-in-Time Solution of Power Systems with Scheduled Events*, PES IEEE, 2018.

XBraid-Adjoint[1] for numerical optimization

- Extend the XBraid interface to accept a user-defined `adjoint-Step()`
  - Solve upper block-bidiagonal adjoint equation
- Automatically generate `adjoint-Step()` with CoDiPack

- Model Problem: Advection-diffusion
  - Minimize difference of space-time averaged solution to preset value

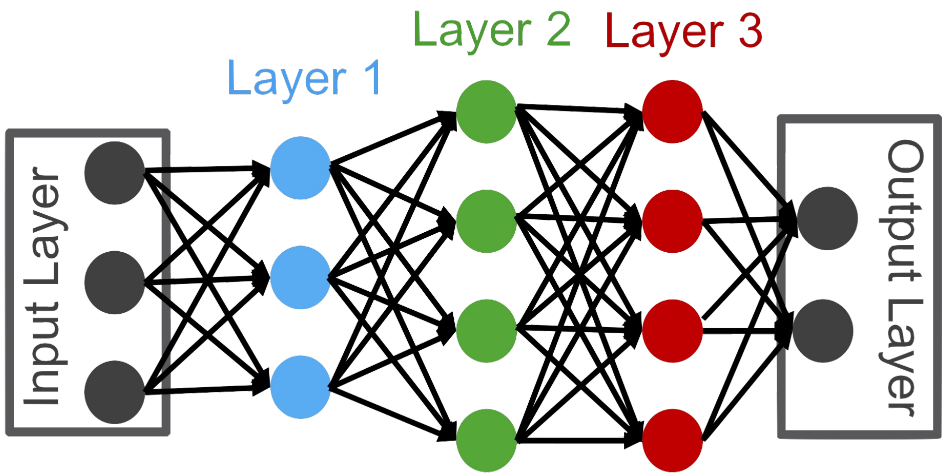
- When used with one-shot strategies, the max speedup is 25x



Scaling of primal (solid lines) and adjoint (dashed lines) XBraid solvers.

1. Günther, Gauger, S., *Non-Intrusive Parallel-in-Time Adjoint Solver with the XBraid Library*, CVS Springer, 2017.

Parallel-in-time and residual neural networks



Insert time-step parameter

- Residual networks (ResNets) ⟷ ODE constrained optimization[1]
  - Let W_n, b_n, y_n be the weights, biases, and state at layer n
  - Classified training with input/output pair: (y_{data}, c_{data})
    - — Forward problem

$$y_0 = y_{data}$$

$$y_{n+1} = y_n + F(W_n y_n + b_n) \quad \forall n = 0, \ldots, N-1$$

    - — Learning problem

$$\min_{W_n, b_n} \text{Loss}(y_N, c_{data}) \quad \text{subject to above forward problem}$$

- Resnet propagation is equivalent to a forward Euler discretization and backpropagation is equivalent to discrete adjoint[1]

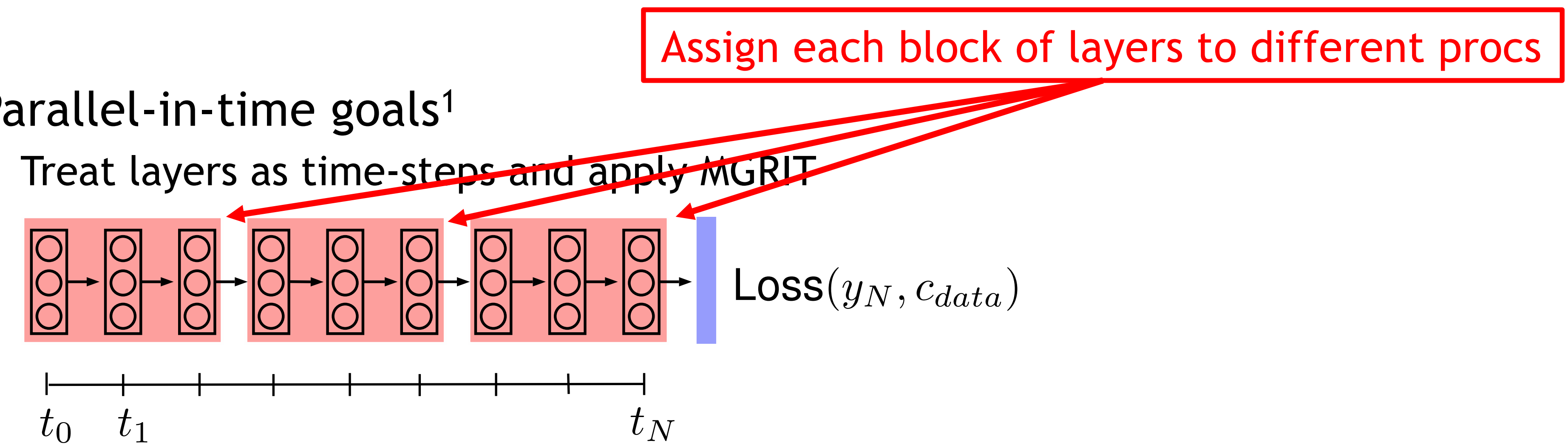
1. Haber, Ruthotto. *Stable Architectures for Deep Neural Networks*. Inverse Probl., 2017.

Parallel-in-time and residual neural networks

- ResNet propagation is equivalent to a forward Euler discretization, and ResNet backpropagation is equivalent to discrete adjoint!

  → *Use this equivalence to apply XBraid-adjoint*

- Parallel-in-time goals[1]
  - Treat layers as time-steps and apply MGRIT

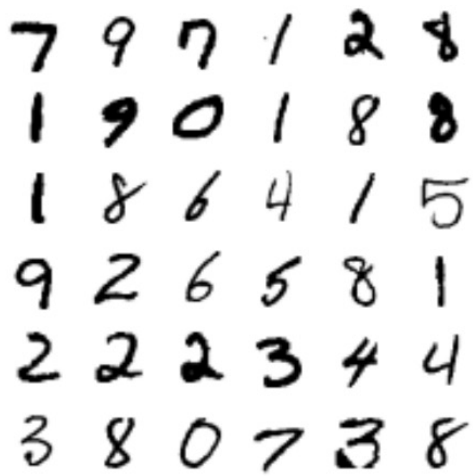


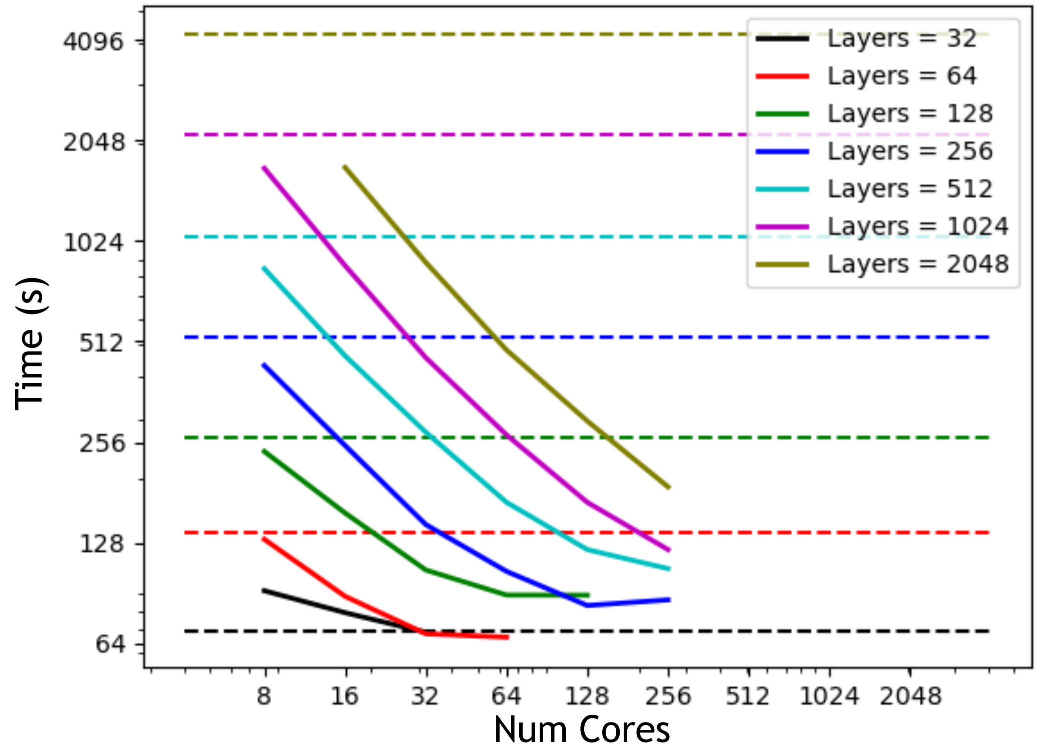
  - Good strong and weak scaling with respect to number of network layers
    → Train a network with 5 layers with same wall-clock time as 1000 layers
  - Solve the same training problem (no shortcuts) as the sequential training version
  - Provide novel layer-parallelism (decoupled layer computations in parallel)

1. Günther, Ruthotto, Schroder, et al. *Parallel-in-Layer Optimization for Training of Deep Residual Networks*. SIMODS, 2020.

Parallel-in-time and neural networks (ResNets)

- Apply XBraid-adjoint solver to ResNet training
  - Goal: Train a network with 5 layers in the same time as 1000 layers
  - Solve the same training problem (no shortcuts) as sequential
- MNIST image classification[1]





Strong scaling for ResNet training
Solid lines: TorchBraid (MGRIT)
Dashed lines: Sequential-in-layer

- Overall, good strong and weak scaling
  - Best training speedup 21x at 4096 layers
    → Yes, it's too many layers, but the point is a scalable algorithm for future problems

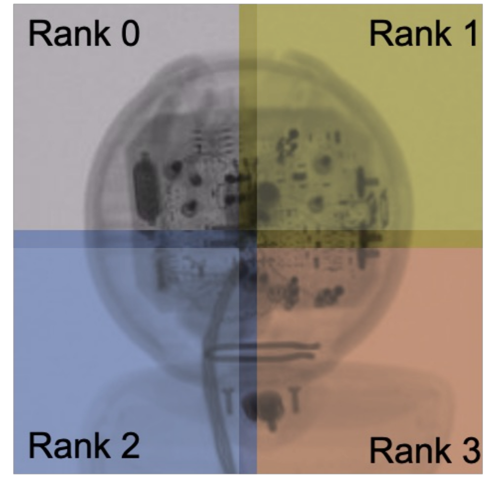
1. Günther, Ruthotto, Schroder, et al. *Parallel-in-Layer Optimization for Training of Deep Residual Networks*. SIMODS, 2020.

Extreme scale machine learning (ML)

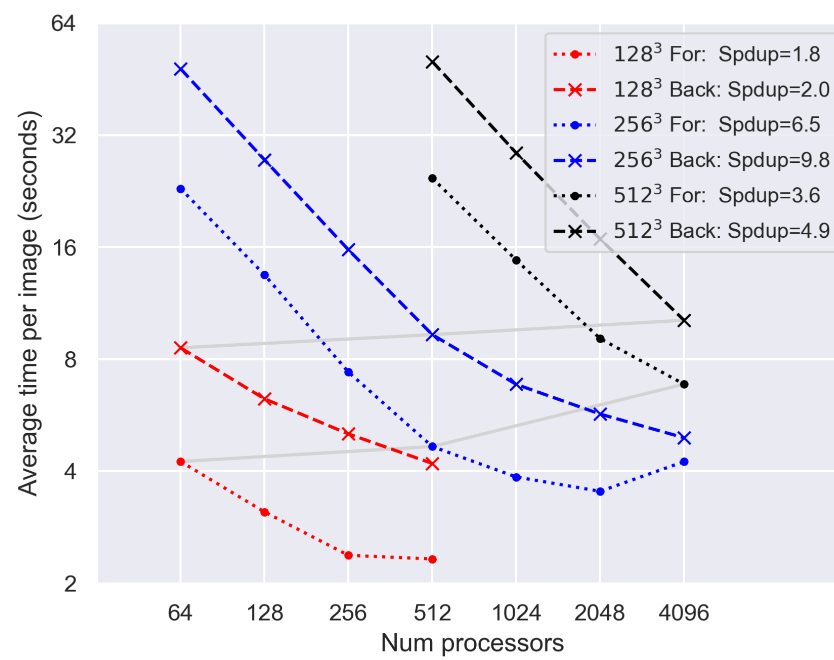
- ML on traditional high-performance supercomputers is an open problem
  - Current work with Hewett, Cyr, & Saavedra
  - Train on 10^3, 10^6, …, compute nodes?
  - Urgently needed for >TB datasets
    → Split data (e.g., image) across processors

- Target problems: Sandia CT scans and NMDID database (UNM) (>*TB in size*)
  - Training enabled by novel spatial decomposition coupled with MGRIT
  - Preliminary results promising

- Future: MG/Opt, GPU extensions, …



Parallel spatial decomposition of CT scan for ML



Speedups from spatial decomposition for image segmentation (identify material)

Machine learning algorithmic and parallel speedups: Multigrid optimization (MGOPT) plus layer-parallel

- ML algorithmic speedups possible with MGOPT (multilevel optimization)[1]
  - Core concept: minimize the objective function on hierarchy of refined networks

$$\min_{W_n, b_n} \text{Loss}(y_N, c_{data})$$

    → ODE perspective provides a natural way to coarsen problems in layer (time)
    → Coarser networks provide parameter updates to finer networks

  - Coarse objective functions have an additional term[1] for consistency (FAS)
    - Let g_H, g_h, W_H be coarse gradient, fine gradient, and coarse weights, resp.,
    - Update coarse objective function with new term: $-\langle g_H - g_h, W_H \rangle$
    - Make coarse and fine objective functions "consistent"

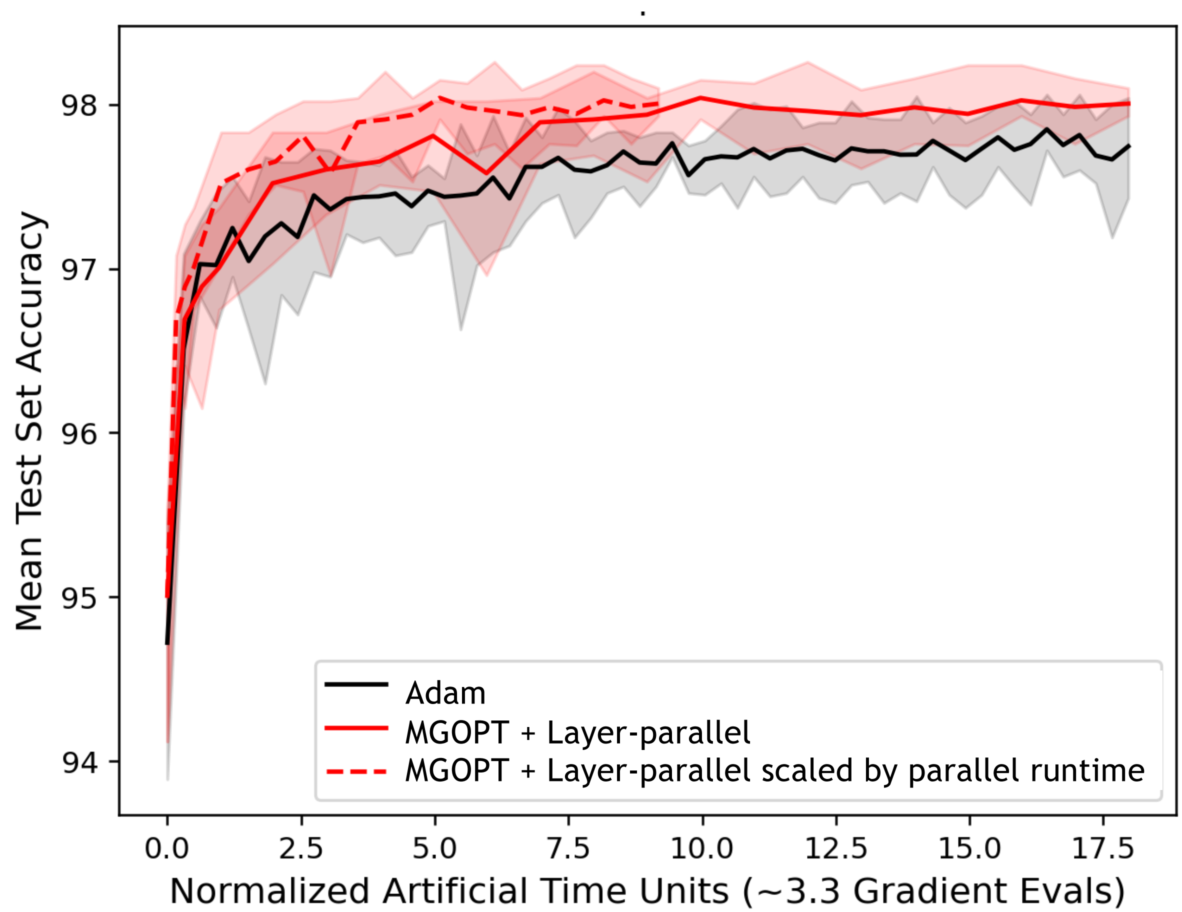
- When applied to ML[2,3] the results are promising and provide an algorithmic speedup for some classification problems

- Can we combine this algorithmic speedup with parallel speedup? Yes!

1. Nash, *A multigrid approach to discretized optimization problems*, Optimization Methods and Software, 2000.
2. von Planta, Kopanicakova, Krause, *Training of deep residual networks with stochastic MG/OPT*, (Arxiv) 2021.
3. Kopanicakova, Krause, *Multilevel minimization for deep residual networks*, (Arxiv) 2020.

Machine learning algorithmic and parallel speedups: Multigrid optimization (MGOPT) plus layer-parallel

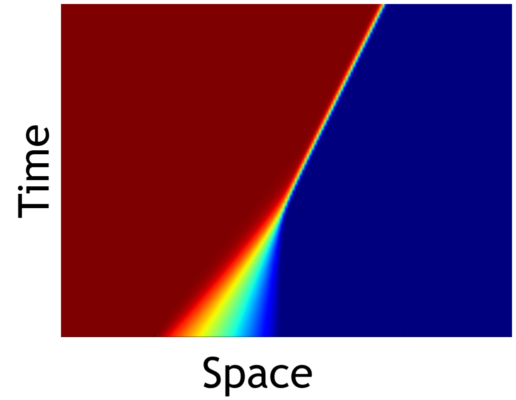
- Train with MNIST
- Adam optimizer versus MGOPT + Layer-parallel
  - Layer-parallel computes gradients for MGOPT
  - 128 layers (as demonstration)
  - Parallel runs on Quartz (Intel cluster at LLNL)
- For this simple problem, MGOPT + layer-parallel exhibits an algorithmic and parallel speedup
- Next: fashion MNIST and other harder problems



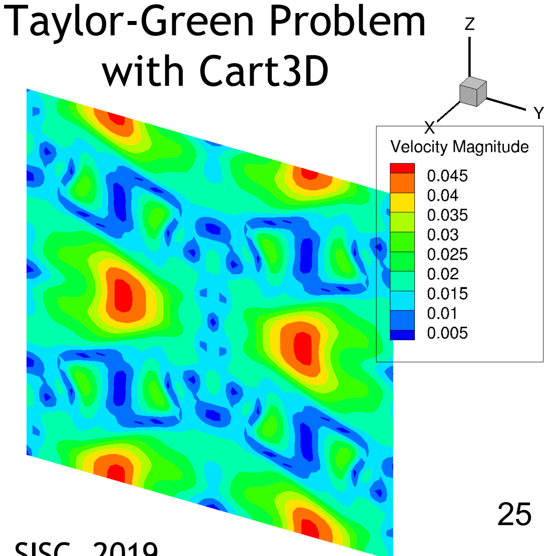
Hyperbolic problems are traditionally difficult

- Important initial successes
- 1D/2D advection and Burgers' equation[1]
  - F-cycles needed (multilevel), slow iteration growth
  - Requires adaptive spatial coarsening
  - Dissipation improves convergence
  - FCF-relaxation and small coarsening factors important
- Recent big improvements for linear advection[2]
  - Special semi-Lagrangian coarse-grid discretization
- Navier-Stokes in 2D and 3D[3,4]
  - Multiple codes: Strand2D, Cart3D, CHeart, Chord
  - Compressible and incompressible, modest *Re*

1D Burgers' Equation



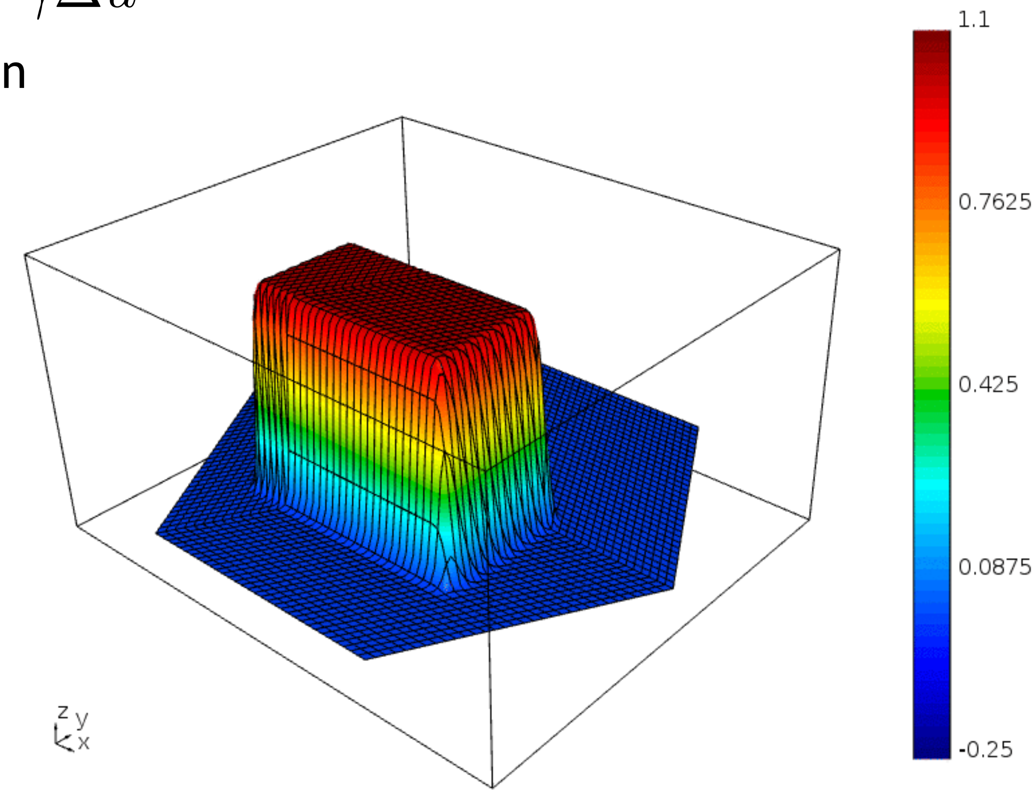
Taylor-Green Problem with Cart3D



1. De Sterck, Howse, Schroder, et al., *Parallel-in-Time MG with Adaptive Coarsening for Inviscid Burgers*, SISC, 2019.
2. Krzysik, De Sterck, Falgout, *Fast MGRIT for Advection via Modified Semi-Lagrangian CG Operators*, 2022, https://arxiv.org/abs/2203.13382
3. Falgout, Katz, Kolev, Schroder, Wissink, Yang, *Parallel Time Integration with MGRIT for Compressible Fluid Dyn.*, 2014.
4. Christopher, Gao, Guzik, Falgout, Schroder, *Space-Time Parallel Alg. with Adaptive Mesh Refinement for CFD*, CVS Springer, 2020.

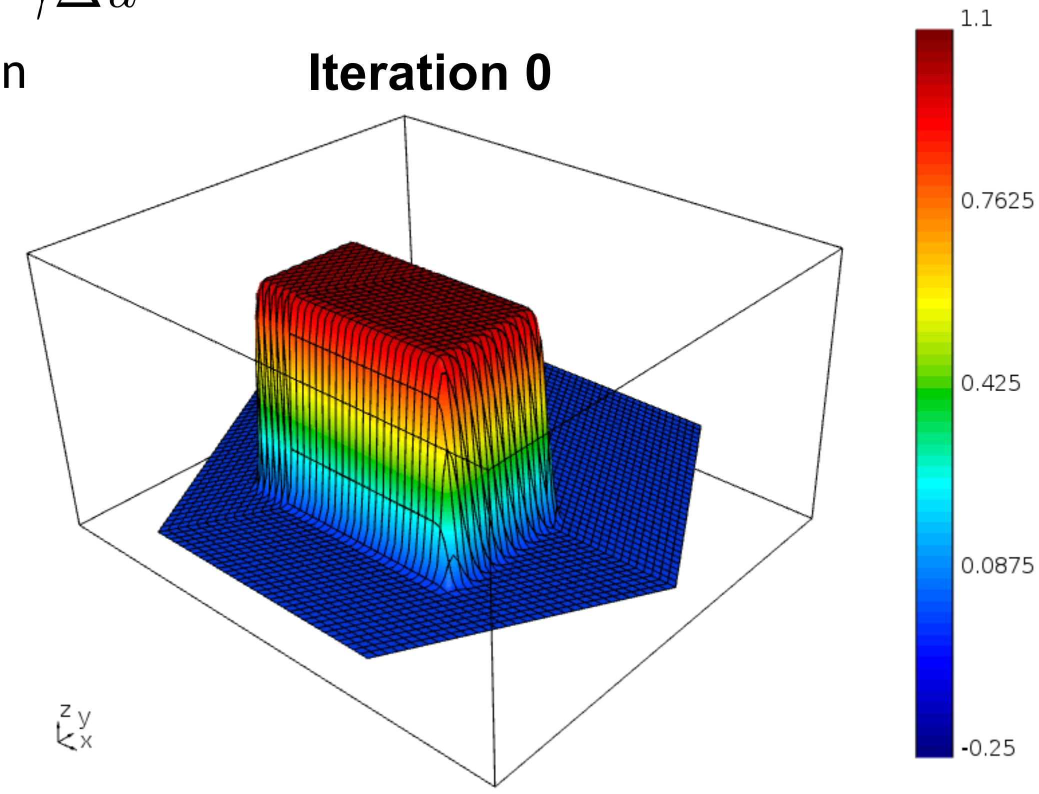
Hyperbolic problems: Explicit methods

- 2D advection $u_t = \mathbf{b}(\mathbf{x}) \cdot \nabla u + \gamma \Delta u$
  - 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



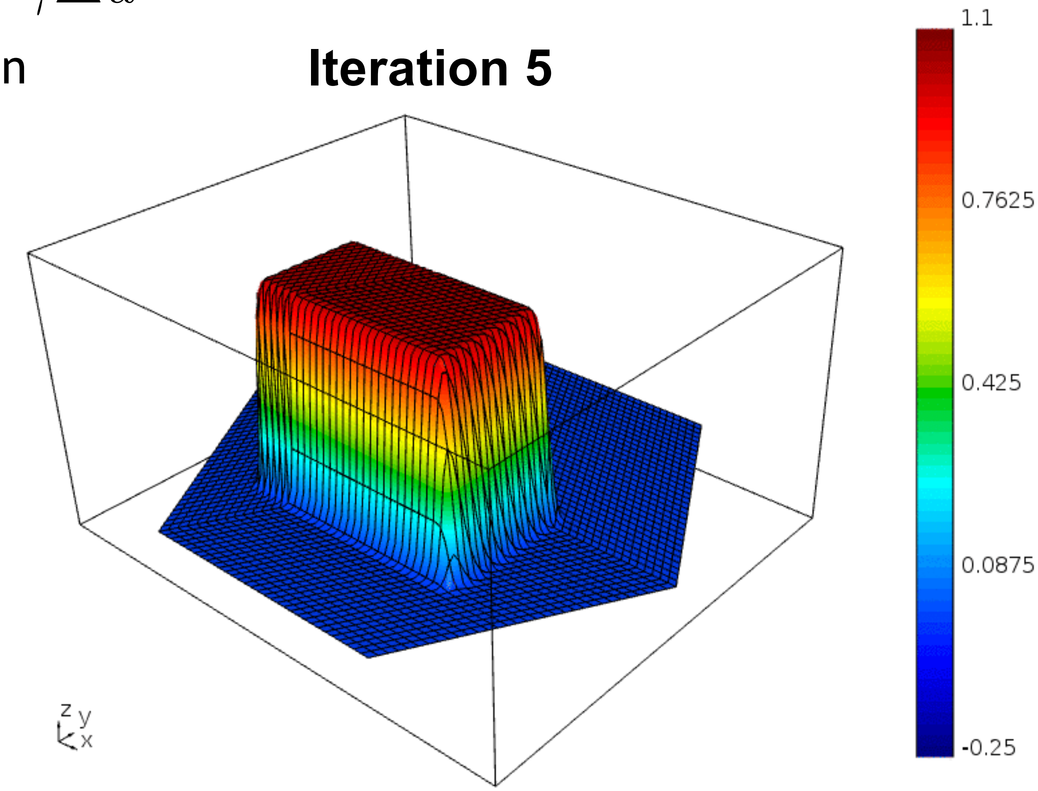
Hyperbolic problems: Explicit methods

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



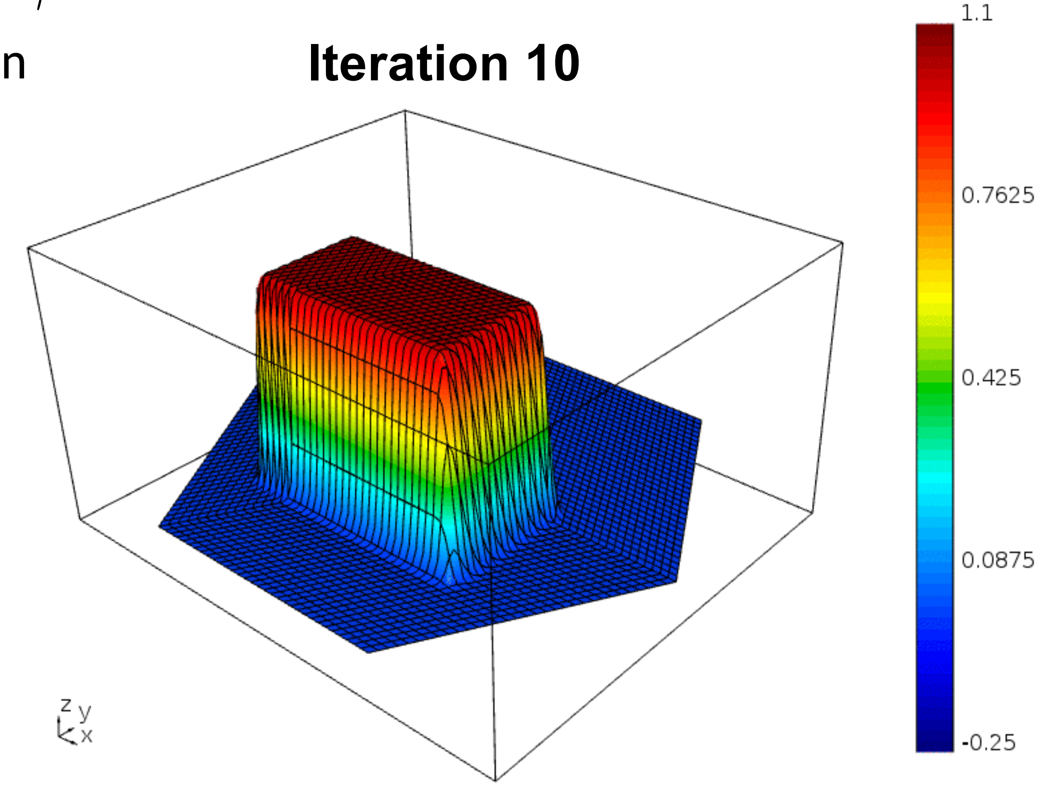
Hyperbolic problems: Explicit methods

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



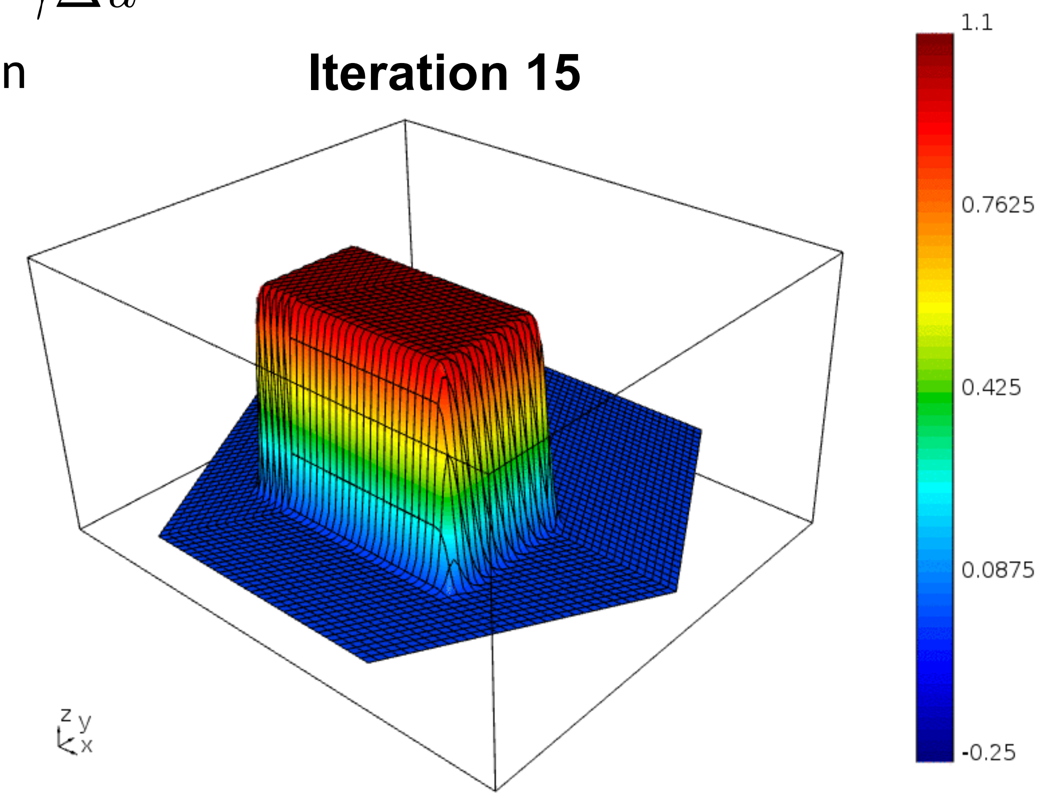
Hyperbolic problems: Explicit methods

- 2D advection $u_t = \mathbf{b}(\mathbf{x}) \cdot \nabla u + \gamma \Delta u$
  - 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



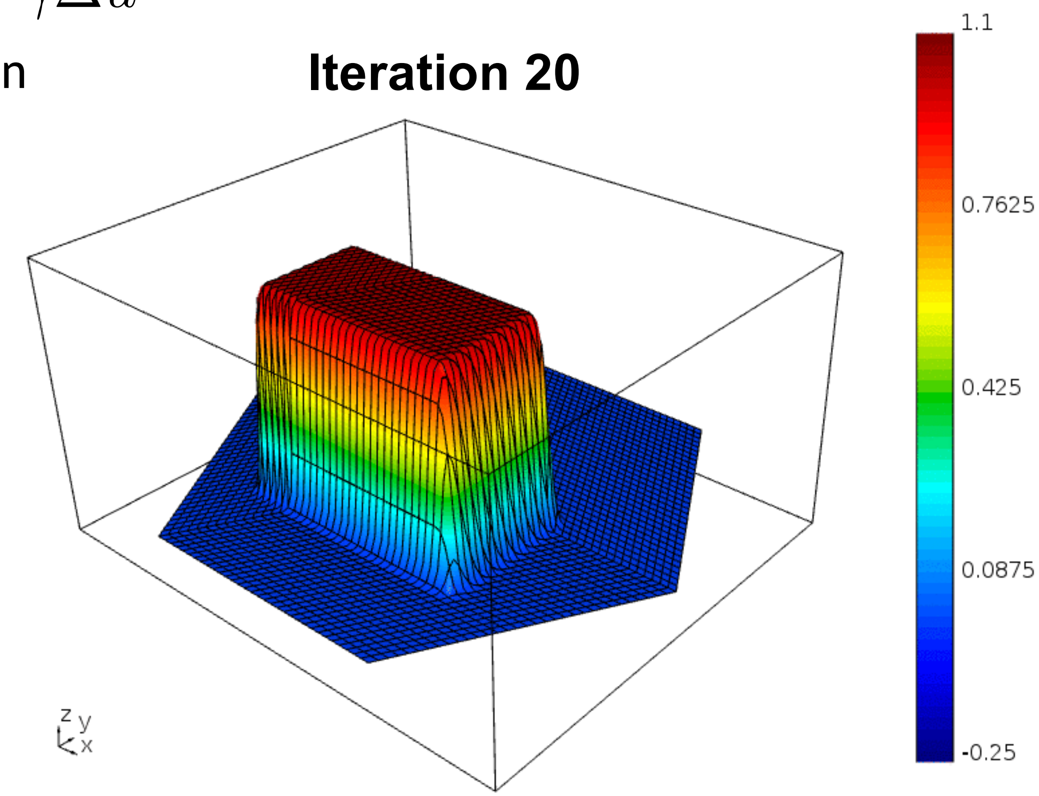
Hyperbolic problems: Explicit methods

- 2D advection $u_t = \mathbf{b}(\mathbf{x}) \cdot \nabla u + \gamma \Delta u$
  - 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



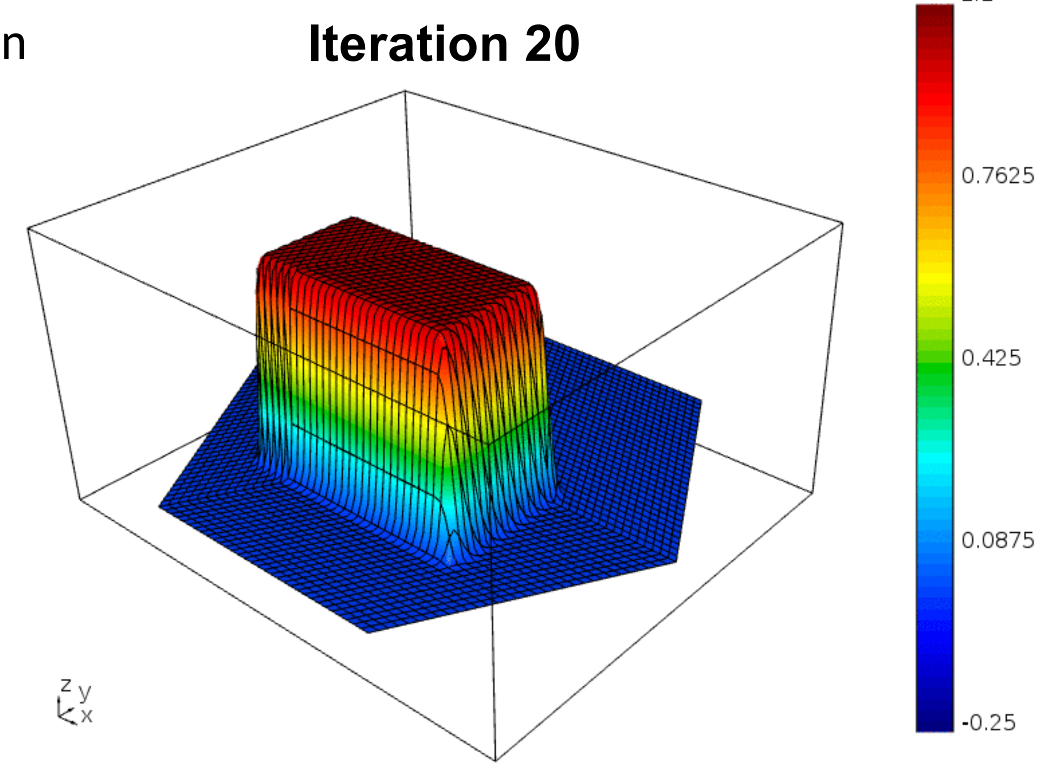
Hyperbolic problems: Explicit methods

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



Hyperbolic problems: Explicit methods

- 2D advection $u_t = \mathbf{b}(\mathbf{x}) \cdot \nabla u + \gamma \Delta u$
  - 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

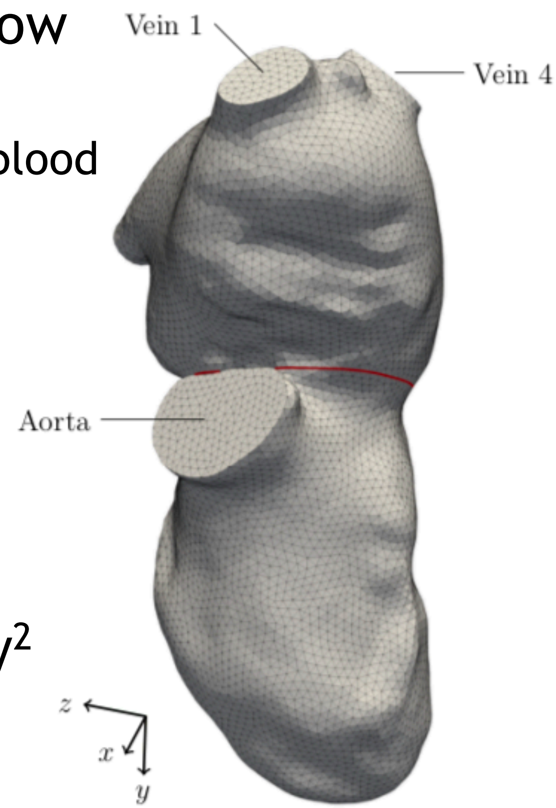


- Future Work
  - Convergence can be **vastly improved** with better coarse-grid equations[1]
  - Consider space-time AMG solvers

1. De Sterck, Falgout, Friedhoff, Krzysik, *Optimizing MGRIT and Parareal coarse-grid operators for linear advection*, NLSAA 2021.

Periodic fluid-structure interaction (FSI)

- Goal: speedup biomedical simulations, e.g., blood flow
  - Example problem: Periodic nonlinear flow in left ventricle
  - Equations: elasticity for solid deformations, Navier-Stokes for blood
- Periodicity allows for greater MGRIT efficiency[1]
  - MGRIT simulates only one periodic time interval
  - Standard method simulates many intervals until steady state
  - 20 processors in time → 5x speedup
- Current research is using multilevel convergence theory[2] to guide algorithm development



Left ventricle

1. Hessenthaler, Falgout, Schroder, Nordsletten, Roehrle, *Time-Periodic Steady-State Solution of Fluid-Structure Interaction and Cardiac Flow Problems through MGRIT*. Comput. Meth. Appl. Mech. Eng., (Submitted) 2021.
2. Hessenthaler, Southworth, Nordsletten, Rohrle, Falgout, Schroder, *Multilevel convergence analysis of MGRIT*, SISC, 2020.

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
  - Hackbusch (1984); Horton (1992); Horton+Vandewalle (1995); Gander+Neumueller (2016)
  - The last two are among the most efficient methods for linear parabolic problems
- 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
  - Minion, Williams (2008, 2010) – PFASST, spectral deferred correction / parareal
  - De Sterck, Manteuffel, McCormick, Olson (2004, 2006) – FOSLS
- Research on these methods is ramping up!
  - Ong, Ruprecht, Krause, Speck, Minion, Langer, De Sterck … 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
  - Enabling novel parallelism in machine learning
  - ...

Selected references

Parallel-in-Time

1. Falgout, Friedhoff, Kolev, MacLachlan, Schroder, *Parallel Time Integration with Multigrid*, SIAM J. Sci. Comput. (SISC), 2014.
2. Dobrev, Kolev, Petersson, Schroder, *Two-level Convergence Theory for MGRIT*, SIAM J. Sci. Comput. (SISC), 2017.
3. Guenther, Ruthotto, Schroder, Cyr, Gauger, *Layer-parallel training of deep residual neural networks*. SIAM J. Math. Data Sci. (SIMODS), 2020.
4. Sugiyama, Schroder, Southworth, Friedhoff, *Weighted Relaxation for Multigrid Reduction in Time*. Numer. Lin. Alg. Appl. Submitted, June 2021.
5. Ong, Schroder, *Applications of Time Parallelization*. CVS, Springer, 2020. *Review paper.*

Software

1. XBraid: https://github.com/XBraid/xbraid

XBRAID
PARALLEL MULTIGRID IN TIME

Outline

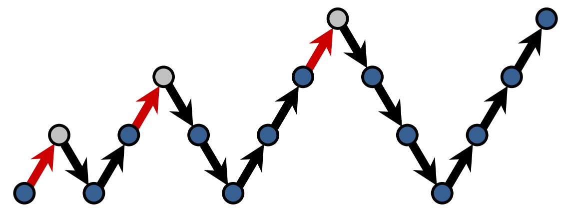
1. Introduction
   → Tutorial software requirements and XBraid overview
2. Simplest example of solving a scalar ODE with `examples/ex-01`
   → Defining the `App` and `vector` structures, writing wrapper functions, running XBraid
3. Explore more XBraid settings in `examples/ex-01-expanded.c`
4. Porting a user-code to XBraid with `examples/ex-02`
   → Debugging the connection to XBraid
   → Intrusiveness versus efficiency
5. A few application area highlights

Appendix: *Advanced XBraid features*

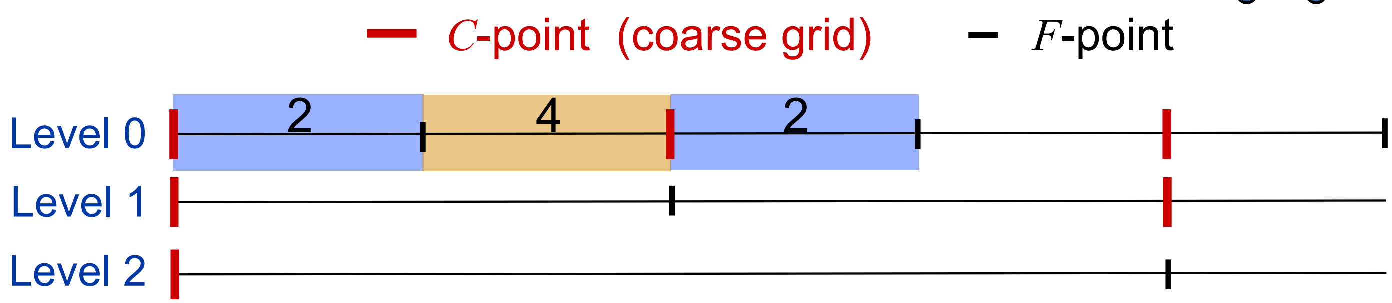
- *Temporal adaptivity*
- *Shell-vectors and BDF-k*
- *Fortran90 Interface*
- *Residual and storage options*
- *Spatial coarsening*
- *Python Interface*

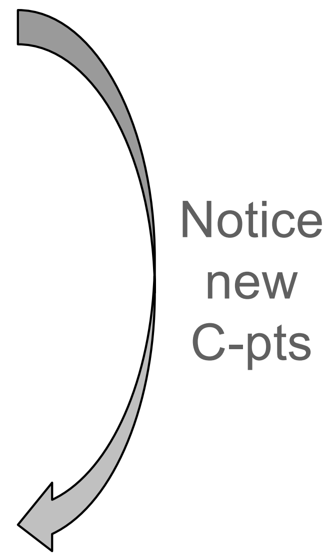
Advanced feature: FMG allows for adaptivity in time and space

- User returns refinement factor in `Step()`

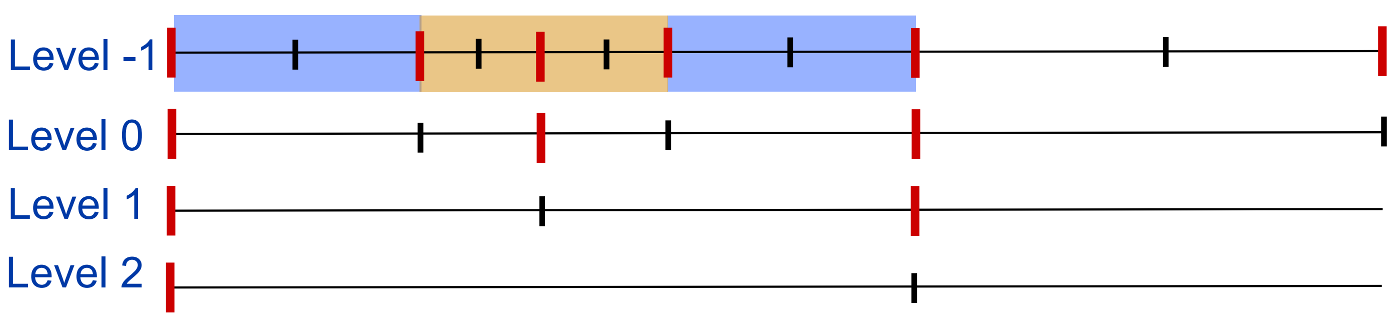


- Example time grid hierarchy





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



Advanced feature: adaptivity in time

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

- **This example uses a built-in Richardson error estimator for refinement in time**
- **`braid_StepStatusSetRFactor(status, k)` refines an interval `k` times**
  - **Called from inside of `Step()`**

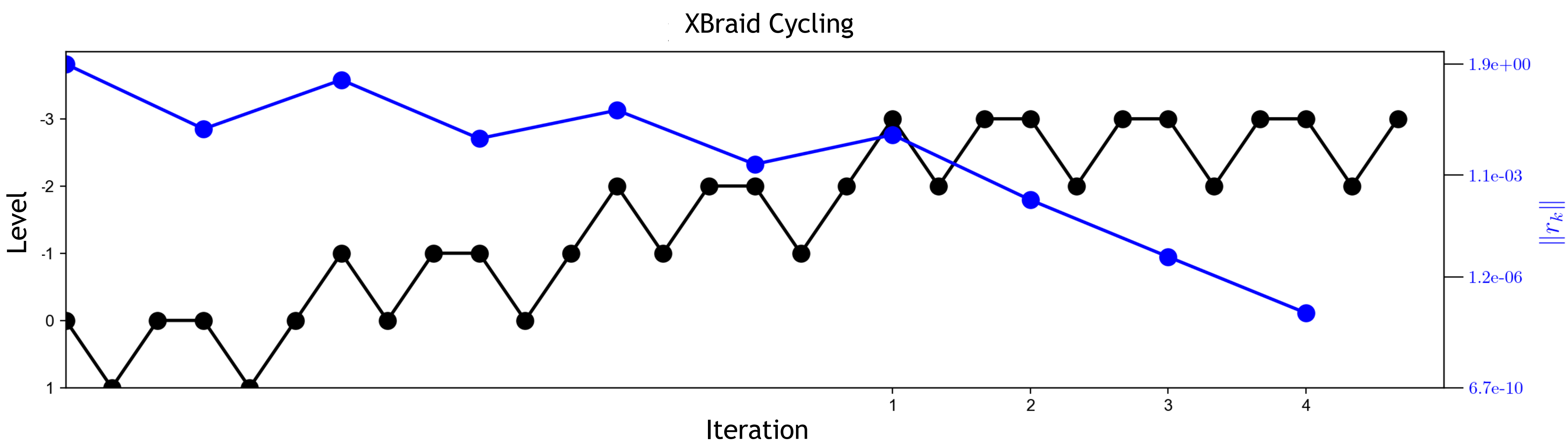
```
$ make ex-02
$ ./ex-02 -ntime 8 -refinet 3e-2

 Braid: Begin simulation, 8 time steps
 Braid: || r_0 || = 1.855448e+00, conv factor = 1.00e+00, wall time = ...
 Braid: || r_1 || = 2.371288e-02, conv factor = 1.28e-02, wall time = ...
 Braid: Temporal refinement occurred, 38 time steps
 Braid: || r_1 || = 6.407304e-01, conv factor = 3.45e-01, wall time = ...
 Braid: || r_2 || = 1.242778e-02, conv factor = 1.94e-02, wall time = ...
 Braid: Temporal refinement occurred, 66 time steps
 Braid: || r_2 || = 8.337944e-02, conv factor = 1.30e-01, wall time = ...
 Braid: || r_3 || = 2.215613e-03, conv factor = 2.66e-02, wall time = ...
 Braid: Temporal refinement occurred, 70 time steps
 Braid: || r_3 || = 1.602040e-02, conv factor = 1.92e-01, wall time = ...
 Braid: || r_4 || = 2.011504e-04, conv factor = 1.26e-02, wall time = ...
 Braid: || r_5 || = 4.412674e-06, conv factor = 2.19e-02, wall time = ...
 Braid: || r_6 || = 1.013677e-07, conv factor = 2.30e-02, wall time = ...
 Discretization error at final time:  2.3758e-02
 ...
```

Advanced feature: adaptivity in time

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

- **Now, visualize the cycling**
- **Observe how the new levels (and time-points) are added**
- **This causes an uneven reduction in the residual**

```
$ python ../misc/user_utils/cycleplot.py
```



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 -nt 128 -nx 9 9 -mi 4 -res
 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

Let space-time block operator be

$$A\mathbf{u} \equiv \begin{pmatrix} I & & & \\ -\Phi & \Psi & & \\ & \ddots & \ddots & \\ & & -\Phi & \Psi \end{pmatrix} \begin{pmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_N \end{pmatrix} = \begin{pmatrix} \mathbf{f}_0 \\ \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_N \end{pmatrix}$$

- Block row of this system: $A_i(\mathbf{u}_i, \mathbf{u}_{i-1}) = f_i$
- Block row of operator: $A_i(\mathbf{u}_i, \mathbf{u}_{i-1}) = -\Phi(\mathbf{u}_{i-1}) + \Psi(\mathbf{u}_i)$
- Residual: $r_i = f_i + A_i(\mathbf{u}_i, \mathbf{u}_{i-1})$

XBraid Default

- User defines `Step(`$\mathbf{u}_{i-1}$`)` $= \Phi(\mathbf{u}_{i-1})$
- XBraid assumes $\Psi = I$
- XBraid computes the residual with no additional information
- BUT for implicit, Φ must be a full implicit solve on finest level for accurate residual
- **OUCH!** This residual computation has same FLOPS as serial time-stepping.
- **Residual setting:** remove this cost
  - Compute the residual with another new user-defined function

Understanding the residual feature

- **Residual setting:** define new user function for cheap residual computation

$$r_i = f_i + A_i(\mathbf{u}_i, \mathbf{u}_{i-1})$$

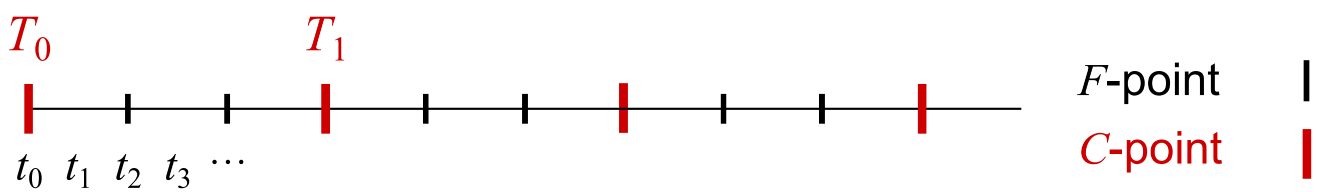
New function

$$r_i = f_i + \texttt{Residual}(\texttt{u}_\texttt{i},\ \texttt{u}_\texttt{i-1})$$

- `Residual(u_i, u_i-1)` $= A_i(\mathbf{u}_i, \mathbf{u}_{i-1}) = -\Phi(\mathbf{u}_{i-1}) + \Psi(\mathbf{u}_i)$
- Let $\Phi = I,\ \Psi =$ sparse matrix inverted by implicit time-stepping
  → Now, residual computation requires NO matrix inverse and is cheap
- `Step()` now computes $\Psi^{-1}(f_i + \Phi(\mathbf{u}_{i-1})) \rightarrow \mathbf{u}_i$
- BUT, this operation is only used for relaxation
  → THUS, cheap inexact solves are used, e.g., 1 or 2 spatial multigrid V-cycles
- Note the f_i term
  - Provided to user with the `fstop` vector in `Step()`
  - This is the forcing term provided by FAS on coarse MGRIT levels

Advanced feature: shell-vectors & BDF-k

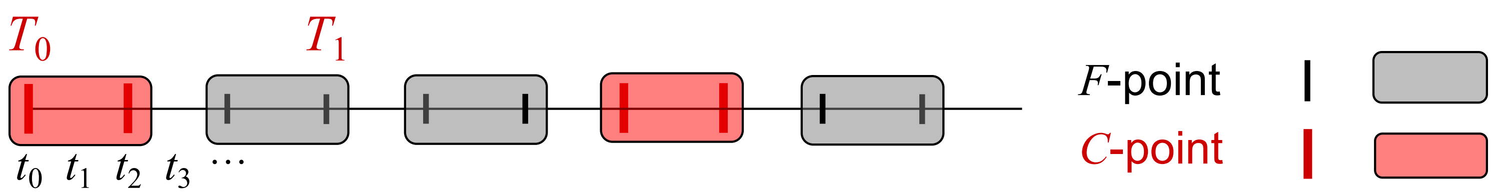
- File: `examples/ex-01-expanded-bdf2.c` Solves: $u_t = \lambda u$
- XBraid is designed for one-step methods. This is the standard way to partition the time-line.



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. The new way to partition so that BDF-k looks "one-step" is to group k time-steps together (here, $k = 2$).



  - 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

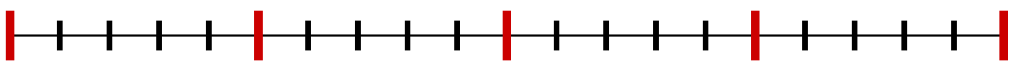
- Other BDF-k strategies, like reducing order on coarse-grids, are possible

- 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***



– F-point (fine grid only)

– C-point (coarse & fine grid)

- **The extra storage critically gives improved initial guesses to implicit solvers**
- **The extra storage changes the problem being solved**
  - **The operator Φ changes as the initial guess changes**

- **Look at the residual histories with**

```
$ make ex-03
$ ./ex-03 -nx 17 17 -nt 128 -storage -1

$ ./ex-03 -nx 17 17 -nt 128 -storage  0

$ ./ex-03 -nx 17 17 -nt 128 -storage  1
```

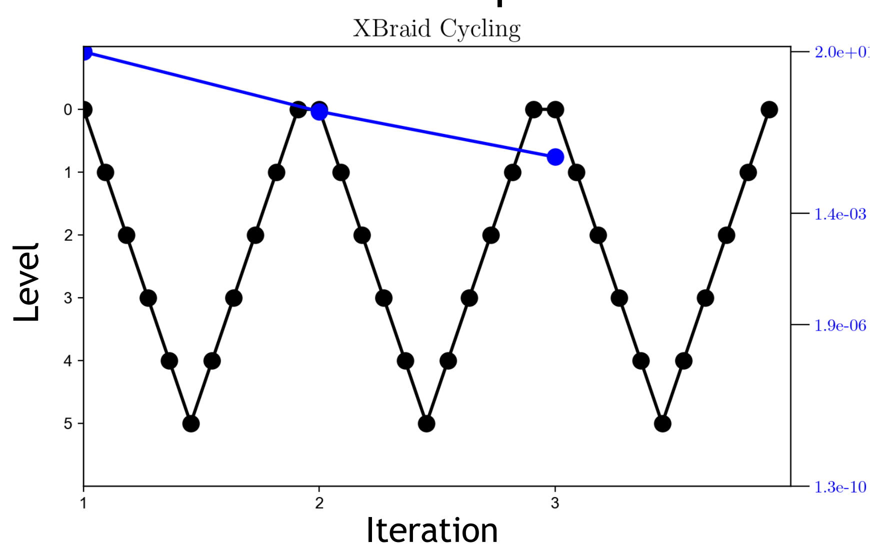
Advanced feature: skip option

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

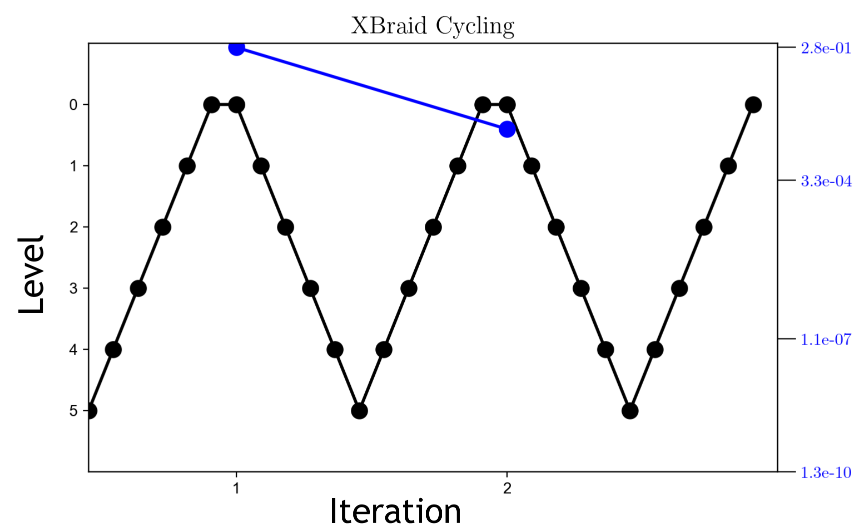
- **Skip allows XBraid to skip (typically useless) relaxations on the 1st down cycle**
  - **By default, skip is turned on**
- **Compare the residual histories for**

```
$ ./ex-03 -nx 17 17 -nt 128 -mi 3 -skip 1
$ ./ex-03 -nx 17 17 -nt 128 -mi 3 -skip 0
```

No Skip



Skip



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

```
$ ./ex-02 -ntime 64 -nspace 17 -ml 3 -sc
  Braid: || r_0 || = 3.652579e+00, conv factor = 1.00e+00, wall time = ...
  Braid: || r_1 || = 1.714767e-01, conv factor = 4.69e-02, wall time = ...
  Braid: || r_2 || = 6.306301e-03, conv factor = 3.68e-02, wall time = ...
  Braid: || r_3 || = 3.238587e-04, conv factor = 5.14e-02, wall time = ...
  ...
  level       dx          dt        dt/dx^2
 -----------------------------------------------------------------
   0   |   1.96e-01    9.82e-02    2.55e+00
   1   |   3.93e-01    1.96e-01    1.27e+00
   2   |   7.85e-01    3.93e-01    6.37e-01

$ ./ex-02 -ntime 64 -nspace 17 -ml 3
  Braid: || r_0 || = 3.652579e+00, conv factor = 1.00e+00, wall time = ...
  Braid: || r_1 || = 1.557155e-01, conv factor = 4.26e-02, wall time = ...
  Braid: || r_2 || = 7.580438e-03, conv factor = 4.87e-02, wall time = ...
  Braid: || r_3 || = 2.430763e-04, conv factor = 3.21e-02, wall time = ...
  ...
  level      dx          dt         dt/dx^2
 -----------------------------------------------------------------
   0   |   1.96e-01    9.82e-02    2.55e+00
   1   |   1.96e-01    1.96e-01    5.09e+00
   2   |   1.96e-01    3.93e-01    1.02e+01
```

Spatial coarsening can 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

```
module braid_types

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

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

```
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
```

Python interface

- File: `examples/ex-01-cython/ex_01.pyx` Solves: $u_t = \lambda u$
- Requires: Cython, MPI4PY, Numpy, Scipy
- Installs with: `ex_01-setup.py` *(see file for instructions)*

User-defined wrapper functions defined in Cython (hybrid Python/C)

```
cdef int my_step(braid_App app, braid_Vector ustop,
                 braid_Vector fstop, braid_Vector u,
                 braid_StepStatus status):
    tstart = 0.0
    tstop = 0.0
    braid_StepStatusGetTstartTstop(status, &tstart, &tstop)

    # Cast C objects as Python objects
    pyU = <PyBraid_Vector> u
    pyApp = <PyBraid_App> app

    pyU.value[0] = 1./(1. + tstop-tstart)*pyU.value[0]
    return 0
```

Python interface

- File: `examples/ex-01-cython/ex_01.pyx` Solves: $u_t = \lambda u$
- Requires: Cython, MPI4PY, Numpy, Scipy
- Installs with: `ex_01-setup.py` *(see file for instructions)*

Run as normal Python package, e.g., with MPI4PY

```
$ mpirun -np K  python3 ex_01_run.py
```

File: `ex_01_run.py`

```
# Use XBraid as normal Python package
import ex_01
core, app = ex_01.InitCoreApp()
ex_01.run_Braid(core, app)
```

Ideas for More Tutorial Examples

- Do more of a Python example
- Add a three-part example [??? Maybe, maybe not...may be full enough]
  - Parareal
  - MGRIT
  - S.t. similar to Gander/Neumueller with PFMG iters and –res, may need to fix code
- Add/Change reaction or convection term to ex-02...? see what happens?
  - Connect to theory for convergence on real, imag, complex eigs