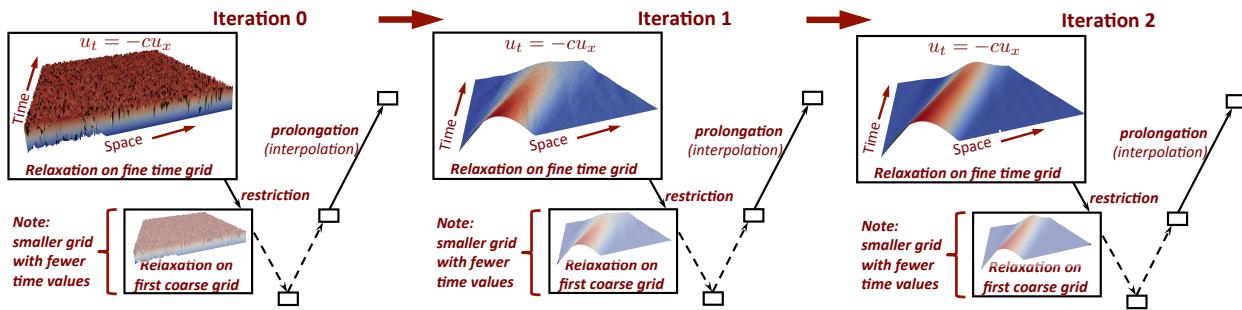


Warp 1.0 Users' Manual



V. A. Dobrev, R. D. Falgout, Tz. V. Kolev,
N. A. Petersson, J. B. Schroder, U. M. Yang,

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-561
Livermore, CA 94551

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

Contents

1 Abstract	1
2 Overview of Warp Algorithm	2
3 A Simple Example	8
4 Building Warp	12
5 Compiling the examples	13
6 Module Index	14
6.1 Modules	14
7 File Index	14
7.1 File List	14
8 Module Documentation	14
8.1 User-written routines	14
8.1.1 Detailed Description	15
8.1.2 Typedef Documentation	15
8.2 User interface routines	17
8.2.1 Detailed Description	17
8.2.2 Typedef Documentation	17
8.2.3 Function Documentation	18
8.3 Warp user test routines	25
8.3.1 Detailed Description	25
8.3.2 Function Documentation	25
9 File Documentation	29
9.1 warp.h File Reference	29
9.1.1 Detailed Description	30
9.1.2 Typedef Documentation	30
9.2 warp_test.h File Reference	30
Index	31

1 Abstract

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

It is **strongly recommended** that you read [Parallel Time Integration with Multigrid](#) before proceeding.

2 Overview of Warp Algorithm

The goal of Warp is to solve a problem faster than is possible with a traditional time marching algorithm. Instead of sequential time marching, Warp solves the problem iteratively by simultaneously updating the current solution guess over all time values. The initial solution guess can be anything, even a random function over space-time. The iterative updates to the solution guess are done by constructing a hierarchy of temporal grids, where the finest grid contains all of the time values for the simulation. Each subsequent grid is a coarser grid with fewer time values. The coarsest grid has a trivial number of time steps and can be quickly solved exactly. The overall effect is that solutions to the time marching problem on the coarser (i.e., cheaper) grids can be used to correct the original finest grid solution.

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

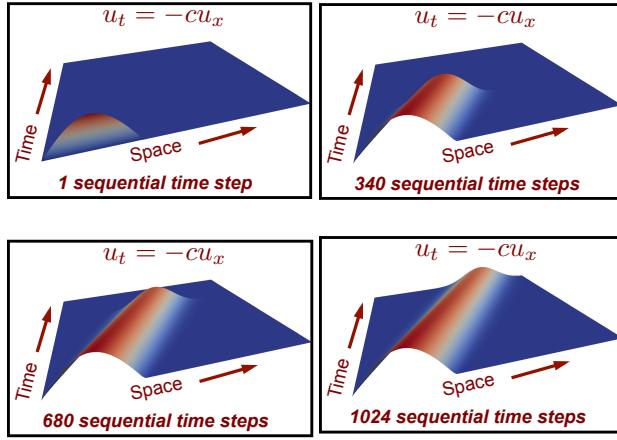


Figure 1: Sequential time stepping.

Warp instead begins with a solution guess over all of space-time, which we let here be random for the sake of argument. A Warp iterations then does

1. Relaxation on the fine grid, i.e., the grid that contains all of the desired time values

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

After the cycle is complete, it repeats until the solution is accurate enough. This is depicted in the next figure, where only a few iterations are required for this simple problem.

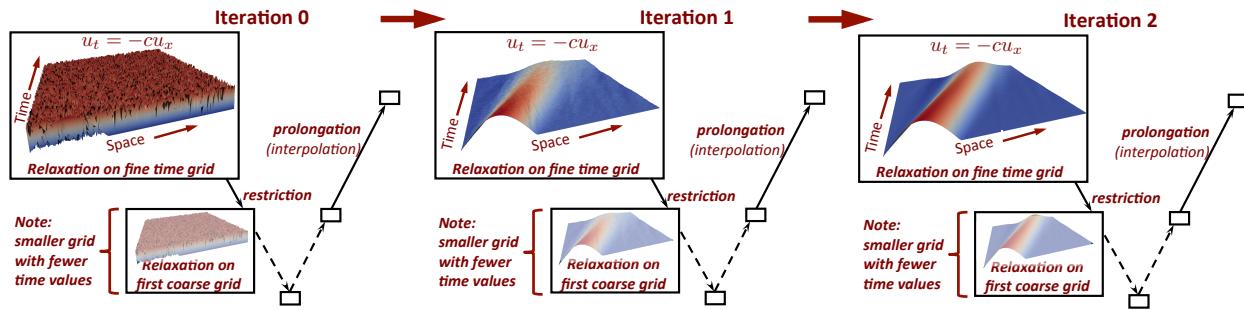


Figure 2: Warp iterations.

There are a few important points to make.

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

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

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

which you discretize with the one-step integration

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

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

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

or

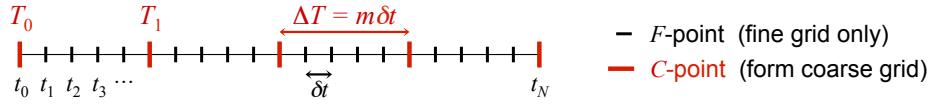
$$A\mathbf{u} = \mathbf{g}.$$

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

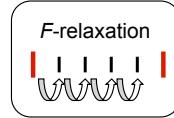
- nonintrusive, in that it coarsens only in time and asks the user to define their own Φ
- optimal and $O(N)$ with a higher constant than time stepping
- highly parallel

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

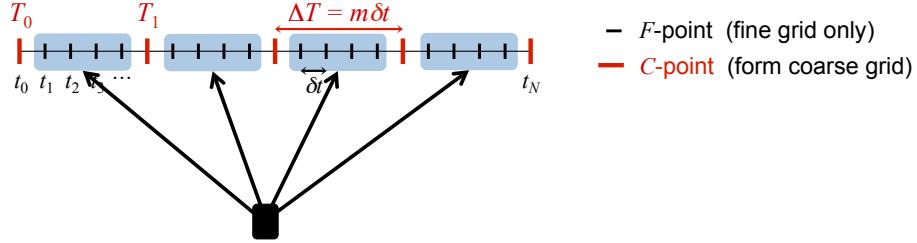
Warp functions as follows. The next Figure depicts a sample timeline of time values, where the time values have been split into C and F points. C points exist on both the fine and coarse time grid, but F points exist only on the fine time scale. Following the above steps outlining a Warp cycle, the first task is relaxation. The simplest relaxation here



alternates between C and F sweeps. An F sweep simply updates time values by integrating with Φ over all the F points from one C point to the next, as depicted here

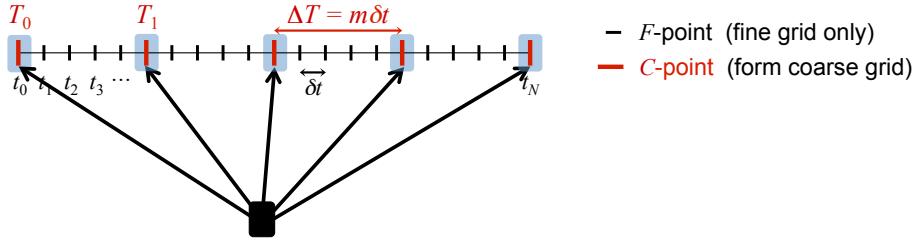


But, such an update can be done simultaneously over all F intervals in parallel, as depicted here



- Update all F-points using time propagator Φ

Following an F sweep we can also do C sweep, as depicted here FCF or F relaxation will refer to how the relaxation



- Update all C-points using time propagator Φ

sweeps are carried out. So, we can say

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

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

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

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

where the spacing between each I is $m - 1$ block rows. This Warp implements an FAS (Full Approximation Scheme) multigrid cycle, and hence the system matrix , solution guess and right-hand-side (i.e., $A, \mathbf{u}, \mathbf{g}$) are all restricted. This is in contrast to linear multigrid which typically restricts the residual equation to the coares grid. We choose FAS because it is nonlinear multigrid and allows us to solve nonlinear problems.

The main question here is how to form the coarse grid matrix, which in turn is defined by the coarse grid time stepper Φ_Δ . It is typical to let Φ_Δ simply be Φ but with the coarse time step size Δt . Thus if

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

then

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

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

$$A_\Delta \mathbf{u}_\Delta = \mathbf{g}_\Delta \equiv R \mathbf{g}$$

is then solved. Finally, \mathbf{u}_Δ is interpolated with P_Φ back to the fine grid, which is equivalent to injecting the coarse grid to the C points on the fine grid, followed by an F relaxation sweep. That is,

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

where m again is the coarsening factor.

This two-grid process is captured with this algorithm. Using a recursive coarse grid solves makes the process multilevel, and halting is done based on a residual tolerance. If the operator is linear, this FAS cycle is equivalent to standard linear multigrid. Note that we represent A as function below, whereas above the notation was simplified for the linear case.

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

In summary, a few points are

- Warp is an iterative solver for the global space-time problem.
- The user defines the time stepping routine Φ .
- Warp convergence will depend heavily on how well Φ_Δ approximates Φ^m , that is how well a time step size of $m\delta t = \Delta T$ will approximate m applications of the same time integrator for a time step size of δt . This is a subject of research, but this approximation need not capture fine scale behavior, which is captured by relaxation on the fine grid.
- The coarsest grid is solved exactly, i.e., sequentially, which is a bottleneck. This is an issue for two-level methods like Parareal,² but not for a multilevel scheme like Warp where the coarsest grid is of trivial size.
- By forming the coarse grid to have the same sparsity structure and time stepper as the fine grid, the algorithm can recur easily and efficiently.
- Interpolation and restriction are ideal or exact, in that an application of interpolation leaves a zero residual at all F points.

Overview of Warp Code

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

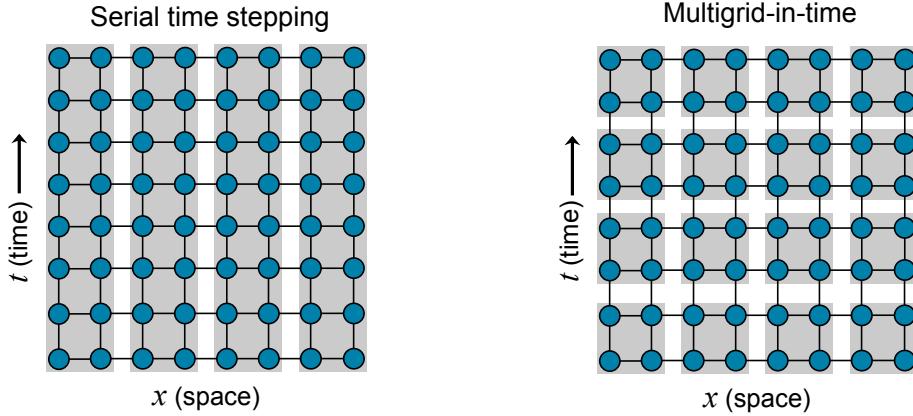
Warp

- is written in C/C++ and can easily interface with Fortran
- uses MPI for parallelism
- self documents through comments in the source code and through *.md files

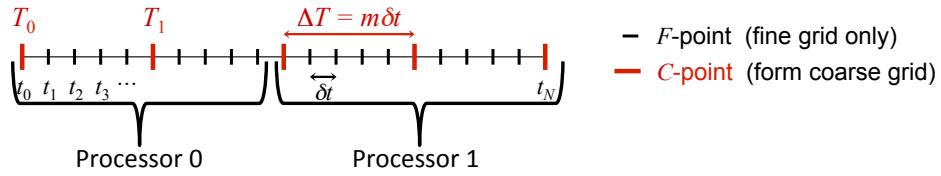
Parallel decomposition and memory

- Warp decomposes the problem in parallel as depicted in this figure. As you can see, traditional time stepping only stores one time step as a time, but only enjoys a spatial data decomposition and spatial parallelism. On the other hand, Warp stores multiple time steps simultaneously and each processor holds a space-time chunk.

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

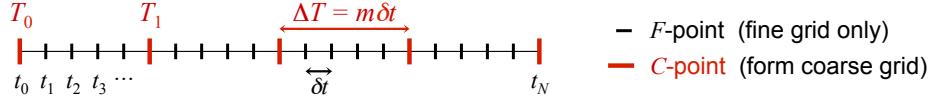


- Warp only handles temporal parallelism and is agnostic to the spatial decomposition. See the [warp_Split-Commworld](#). Each processor owns a certain number of CF intervals of points, as depicted here One interval



is a C point and all following F points until the next C point. These intervals are distributed evenly on the finest grid.

- Storage is greatly minimized by only stored C points. Whenever an F point is needed, it is generated by F relaxation. That is, we only store the red C point time values. Coarsening can be aggressive with $m = 8, 16, 32,$

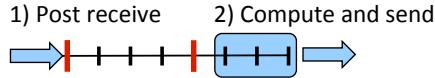


so the storage requirements of Warp are significantly reduced when compared to storing all of the time values.

- In practice, storing only one space-time slab is advisable. That is, solve for as many time steps (say k time steps) as you have available memory for. Then move on to the next k time steps.

Overlapping communication and computation

Warp effectively overlaps communication and computation. The main computation kernel of Warp is an F relaxation sweep, and each processor first posts a send at its left-most C point, and then carries out F relaxation on its right-most interval. If each processor has multiple intervals at this Warp level, this should allow for complete overlap.

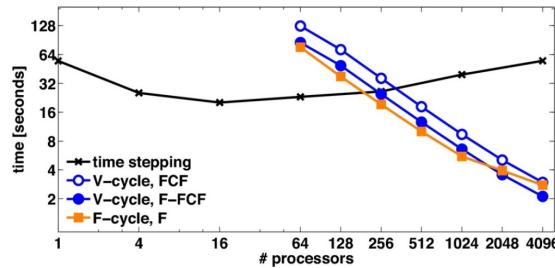


Properties of Warp

- Warp applies multigrid to the time dimension

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

Here is some experimental data for the 2D heat equation generated by the drive-02 example. Here, the speedup is



10x on a Linux cluster using 4 cores per node, a Sandybridge Intel chipset, and a fast Infiniband interconnect. The problem size was $129^2 \times 16,192$. The coarsening factor $m = 16$ on the finest level and 2 on coarser levels. And various relaxation and V and F cycling strategies are experimented with.

3 A Simple Example

User Defined Structures and Wrappers

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

The two data structures are:

1. **App:** This holds a wide variety of information and is `global` in that it is passed to every function. This structure holds everything that the user will need to carry out a simulation. Here, this is just the global MPI communicator and few values describing the temporal domain.

```
typedef struct _warp_App_struct
{
    MPI_Comm    comm;
    double      tstart;
    double      tstop;
    int         ntime;

} my_App;
```

2. **Vector:** this defines something like a state vector at a certain time value. It could contain any other information related to this vector needed to evolve the vector to the next time value, like mesh information.

```
typedef struct _warp_Vector_struct
{
    double      value;

} my_Vector;
```

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

1. **Phi:** this is the time stepping routine. The user must advance the vector u from time $tstart$ to time $tstop$. Here advancing the solution just involves the scalar λ . The `rfactor_ptr` and accuracy inputs are advanced topics.

Importantly, the g_i function (from the Introduction) must be incorporated into `Phi`, so that $\text{Phi}(u_i) \rightarrow u_{i+1}$

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

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

    /* no refinement */
    *rfactor_ptr = 1;

    return 0;
}
```

1. **Init:** initializes a vector at time t . Here that is just allocating and setting a scalar on the heap.

```
int
my_Init(warp_App      app,
         double        t,
         warp_Vector *u_ptr)
{
    my_Vector *u;

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

    return 0;
}
```

2. **Clone:** clones a vector into a new vector

```
int
my_Clone(warp_App      app,
          warp_Vector  u,
          warp_Vector *v_ptr)
{
    my_Vector *v;

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

    return 0;
}
```

3. **Free:** frees a vector

```
int
my_Free(warp_App      app,
         warp_Vector u)
```

```
{
    free(u);
    return 0;
}
```

4. Sum: sums two vectors (AXPY operation)

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

    return 0;
}
```

5. Dot: takes the dot product of two vectors

```
int
my_Dot(warp_App    app,
        warp_Vector u,
        warp_Vector v,
        double      *dot_ptr)
{
    double dot;

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

    return 0;
}
```

6. Write: writes a vector to screen, file, etc... The user defines what is appropriate output. Notice how you are told the time value of the vector *u* and even more information in status. This lets you tailor the output to only outputting certain time values and even from various Warp levels and Warp iterations if status is queried.

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

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

    MPI_Comm_rank(comm, &myid);

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

    return 0;
}
```

7. BufSize, BufPack, BufUnpack: packs a vector into a `void *` buffer for MPI and then unpacks it from `void *` to vector. Here doing that for a scalar is trivial. BufSize computes the upper bound for the size of an arbitrary vector.

```
int
my_BufSize(warp_App    app,
            int       *size_ptr)
```

```

{
    *size_ptr = sizeof(double);
    return 0;
}

int
my_BufPack(warp_App      app,
            warp_Vector   u,
            void          *buffer)
{
    double *dbuffer = buffer;

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

    return 0;
}

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

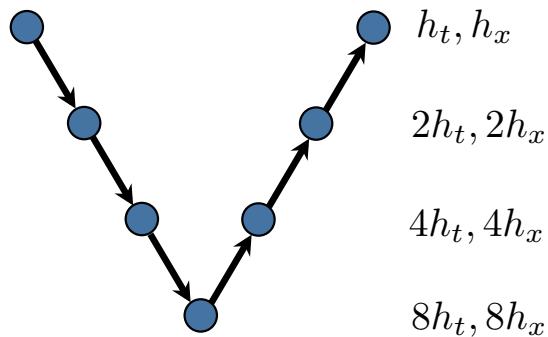
    u = (my_Vector *) malloc(sizeof(my_Vector));
    (u->value) = dbuffer[0];
    *u_ptr = u;

    return 0;
}

```

8. **Coarsen, Restrict** (optional): advanced option that allows for coarsening in space while you coarsen in time. This is useful for maintaining stable explicit schemes on coarse time scales and is not needed here. See for instance drive-04 and drive-05 which use these routines.

These functions allow you vary the spatial mesh size on Warp levels as depicted here



9. Adaptive, variable time stepping is in the works to be implemented. The rfactor parameter in Phi will allow this.

Running Warp

A typical flow in the `main` function to use Warp is to first initialize the app structure

```

/* set up app structure */
app = (my_App *) malloc(sizeof(my_App));
(app->comm)   = comm;
(app->tstart) = tstart;
(app->tstop)  = tstop;
(app->nitime) = nitime;

```

then the data structure definitions and wrapper routines are passed to Warp

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

then Warp options can be set

```
warp_SetPrintLevel( core, 1);
warp_SetMaxLevels(core, max_levels);
warp_SetNRelax(core, -1, nrelax);
warp_SetAbsTol(core, tol);
warp_SetCFactor(core, -1, cfactor);
warp_SetMaxIter(core, max_iter);
```

then the simulation is run

```
warp_Drive(core);
```

then we clean up

```
warp_Destroy(core);
```

see drive-01 and drive-0* for more extensive examples.

Testing Warp

The best overall test for Warp, is to set MaxLevels to 1 which will carry out a sequential time stepping test. Take the output given to you by your Write function and compare it to output from a non-Warp run. Is everything OK? Repeat this test for multilevel Warp.

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

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

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

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

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

/* Test bufsize(), bufpack(), bufunpack() */
correct = warp_TestBuf( app, comm_x, stdout, dt, my_Init, my_Free, my_Sum, my_Dot, my_BufSize, my_BufPack, my_BufUnpack);

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

4 Building Warp

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

5 Compiling the examples

- Type
`drive-0* -help`

for instructions on how to run any driver

- To run the examples
`mpirun -np 4 drive-* [args]`
- drive-01 is the simplest example. It implements a scalar ODE and can be compiled and run with no outside dependencies.
- drive-02 implements the 2D heat equation on a regular grid. You must have `hypre` installed and these variables in the Makefile set correctly
`HYPRE_DIR = ../../linear_solvers/hypre`
`HYPRE_FLAGS = -I$(HYPRE_DIR)/include`
`HYPRE_LIB = -L$(HYPRE_DIR)/lib -lHYPRE`
- drive-03 implements the 3D heat equation on a regular grid, and assumes hypre is installed just like drive-02.
- drive-05 implements the 2D heat equation on a regular grid, but it uses with spatial coarsening. This allows you to use explicit time stepping on each Warp level, regardless of time step size. It assumes hypre is installed just like drive-02.
- drive-04 is sophisticated test bed for various PDEs, mostly parabolic. It relies on the `mfem` package to create general finite element discretizations for the spatial problem. Other packages must be installed in this order.
 - Unpack and install `Metis`
 - Unpack and install `hypre`
 - Unpack and install `mfem`. Make the serial version of `mfem` first by only typing `make`. Then make sure to set these variables correctly in the Makefile
`USE_METIS_5 = YES HYPRE_DIR = where_ever_linear_solvers_is/hypre`
 - Make `GLVIS`, which needs serial `mfem`.
 - Go back to `mfem`
`make clean`
`make parallel`
 - Got to warp/examples and set these Makefile variables, and them make `drive-04`.

```

METIS_DIR = ../../metis-5.1.0/lib
MFEM_DIR = ../../mfem
MFEM_FLAGS = -I$(MFEM_DIR)
MFEM_LIB = -L$(MFEM_DIR) -lmfem -L$(METIS_DIR) -lmetis

• To run drive-04 and glvis, open two windows. In one, start a glvis session
    ./glvis
Then, in the other window, run drive-04
    mpirun -np ... drive-04 [args]

```

6 Module Index

6.1 Modules

Here is a list of all modules:

User-written routines	14
User interface routines	17
Warp user test routines	25

7 File Index

7.1 File List

Here is a list of all files with brief descriptions:

warp.h Define headers for user interface routines	29
warp_test.h	30

8 Module Documentation

8.1 User-written routines

Typedefs

- `typedef struct _warp_App_struct * warp_App`
- `typedef struct _warp_Vector_struct * warp_Vector`
- `typedef struct _warp_Status_struct * warp_Status`
- `typedef warp_Int(* warp_PtFcnPhi)(warp_App app, warp_Real tstart, warp_Real tstop, warp_Real accuracy, warp_Vector u, warp_Int *rfactor_ptr)`
- `typedef warp_Int(* warp_PtFcnInit)(warp_App app, warp_Real t, warp_Vector *u_ptr)`
- `typedef warp_Int(* warp_PtFcnClone)(warp_App app, warp_Vector u, warp_Vector *v_ptr)`
- `typedef warp_Int(* warp_PtFcnFree)(warp_App app, warp_Vector u)`
- `typedef warp_Int(* warp_PtFcnSum)(warp_App app, warp_Real alpha, warp_Vector x, warp_Real beta, warp_Vector y)`

- `typedef warp_Int(* warp_PtFcnDot)(warp_App app, warp_Vector u, warp_Vector v, warp_Real *dot_ptr)`
- `typedef warp_Int(* warp_PtFcnWrite)(warp_App app, warp_Real t, warp_Status status, warp_Vector u)`
- `typedef warp_Int(* warp_PtFcnBufSize)(warp_App app, warp_Int *size_ptr)`
- `typedef warp_Int(* warp_PtFcnBufPack)(warp_App app, warp_Vector u, void *buffer)`
- `typedef warp_Int(* warp_PtFcnBufUnpack)(warp_App app, void *buffer, warp_Vector *u_ptr)`
- `typedef warp_Int(* warp_PtFcnCoarsen)(warp_App app, warp_Real tstart, warp_Real f_tminus, warp_Real f_tplus, warp_Real c_tminus, warp_Real c_tplus, warp_Vector fu, warp_Vector *cu_ptr)`
- `typedef warp_Int(* warp_PtFcnRefine)(warp_App app, warp_Real tstart, warp_Real f_tminus, warp_Real f_tplus, warp_Real c_tminus, warp_Real c_tplus, warp_Vector cu, warp_Vector *fu_ptr)`

8.1.1 Detailed Description

These are all user-written data structures and routines

8.1.2 Typedef Documentation

8.1.2.1 `typedef struct _warp_App_struct* warp_App`

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

8.1.2.2 `typedef warp_Int(* warp_PtFcnBufPack)(warp_App app,warp_Vector u,void *buffer)`

Packs a vector u into a void * buffer for MPI

8.1.2.3 `typedef warp_Int(* warp_PtFcnBufSize)(warp_App app,warp_Int *size_ptr)`

Computes an upper bound for the MPI Buffer size in bytes for an arbitrary warp_Vector

8.1.2.4 `typedef warp_Int(* warp_PtFcnBufUnpack)(warp_App app,void *buffer,warp_Vector *u_ptr)`

Unpack that buffer from void * to warp_Vector

8.1.2.5 `typedef warp_Int(* warp_PtFcnClone)(warp_App app,warp_Vector u,warp_Vector *v_ptr)`

Clone u into v_ptr

8.1.2.6 `typedef warp_Int(* warp_PtFcnCoarsen)(warp_App app,warp_Real tstart,warp_Real f_tminus,warp_Real f_tplus,warp_Real c_tminus,warp_Real c_tplus,warp_Vector fu,warp_Vector *cu_ptr)`

Spatial coarsening (optional). Allows the user to coarsen when going from a fine time grid to a coarse time grid. This function is called on every vector at each level, thus you can coarsen the entire space time domain.

8.1.2.7 `typedef warp_Int(* warp_PtFcnDot)(warp_App app,warp_Vector u,warp_Vector v,warp_Real *dot_ptr)`

Carry out a dot product *dot_ptr = $\langle u, v \rangle$

8.1.2.8 `typedef warp_Int(* warp_PtFcnFree)(warp_App app,warp_Vector u)`

Free and deallocate u

8.1.2.9 `typedef warp_Int(* warp_PtFcnInit)(warp_App app,warp_Real t,warp_Vector *u_ptr)`

Initializes a vector at time t

8.1.2.10 `typedef warp_Int(* warp_PtFcnPhi)(warp_App app,warp_Real tstart,warp_Real tstop,warp_Real accuracy,warp_Vector u,warp_Int *rfactor_ptr)`

Defines the central time stepping function that the user must write. The user must advance the vector u from time $tstart$ to time $tstop$. Here advancing the solution just involves the scalar λ . The `rfactor_ptr` and `accuracy` inputs are advanced topics. `rfactor_ptr` allows the user to tell Warp to refine this time interval.

8.1.2.11 `typedef warp_Int(* warp_PtFcnRefine)(warp_App app,warp_Real tstart,warp_Real f_tminus,warp_Real f_tplus,warp_Real c_tminus,warp_Real c_tplus,warp_Vector cu,warp_Vector *fu_ptr)`

Spatial refinement (optional). Allows the user to refine when going from a coarse time grid to a fine time grid. This function is called on every vector at each level, thus you can refine the entire space time domain.

8.1.2.12 `typedef warp_Int(* warp_PtFcnSum)(warp_App app,warp_Real alpha,warp_Vector x,warp_Real beta,warp_Vector y)`

$AXPY, \alpha*x + \beta*y \rightarrow y$

8.1.2.13 `typedef warp_Int(* warp_PtFcnWrite)(warp_App app,warp_Real t,warp_Status status,warp_Vector u)`

Write the vector u at time t . The user decides whether to write to file, screen or do nothing. Only solution values at certain times or Warp iterations may be desired.

8.1.2.14 `typedef struct _warp_Status_struct* warp_Status`

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

8.1.2.15 `typedef struct _warp_Vector_struct* warp_Vector`

This defines something like a state vector at a certain time value. It could contain any other information related to this vector needed to evolve the vector to the next time value, like mesh information.

8.2 User interface routines

TypeDefs

- `typedef struct _warp_Core_struct * warp_Core`

Functions

- `warp_Int warp_Init (MPI_Comm comm_world, MPI_Comm comm, warp_Real tstart, warp_Real tstop, warp_Int ntime, warp_App app, warp_PtFcnPhi phi, warp_PtFcnInit init, warp_PtFcnClone clone, warp_PtFcnFree free, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnWrite write, warp_PtFcnBufSize bufsize, warp_PtFcnBufPack bufpack, warp_PtFcnBufUnpack bufunpack, warp_Core *core_ptr)`
- `warp_Int warp_Drive (warp_Core core)`
- `warp_Int warp_Destroy (warp_Core core)`
- `warp_Int warp_PrintStats (warp_Core core)`
- `warp_Int warp_SetLoosexTol (warp_Core core, warp_Int level, warp_Real loose_tol)`
- `warp_Int warp_SetTightxTol (warp_Core core, warp_Int level, warp_Real tight_tol)`
- `warp_Int warp_SetMaxLevels (warp_Core core, warp_Int max_levels)`
- `warp_Int warp_SetMaxCoarse (warp_Core core, warp_Int max_coarse)`
- `warp_Int warp_SetAbsTol (warp_Core core, warp_Real atol)`
- `warp_Int warp_SetRelTol (warp_Core core, warp_Real rtol)`
- `warp_Int warp_SetNRelax (warp_Core core, warp_Int level, warp_Int nrelax)`
- `warp_Int warp_SetCFactor (warp_Core core, warp_Int level, warp_Int cfactor)`
- `warp_Int warp_SetMaxIter (warp_Core core, warp_Int max_iter)`
- `warp_Int warp_SetFMG (warp_Core core)`
- `warp_Int warp_SetNFMGVcyc (warp_Core core, warp_Int nfmvg_Vcyc)`
- `warp_Int warp_SetSpatialCoarsen (warp_Core core, warp_PtFcnCoarsen coarsen)`
- `warp_Int warp_SetSpatialRefine (warp_Core core, warp_PtFcnRefine refine)`
- `warp_Int warp_SetPrintLevel (warp_Core core, warp_Int print_level)`
- `warp_Int warp_SetPrintFile (warp_Core core, const char *printfile_name)`
- `warp_Int warp_SetWriteLevel (warp_Core core, warp_Int write_level)`
- `warp_Int warp_SplitCommworld (const MPI_Comm *comm_world, warp_Int px, MPI_Comm *comm_x, MPI_Comm *comm_t)`
- `warp_Int warp_GetStatusResidual (warp_Status status, warp_Real *rnorm_ptr)`
- `warp_Int warp_GetStatusIter (warp_Status status, warp_Int *iter_ptr)`
- `warp_Int warp_GetStatusLevel (warp_Status status, warp_Int *level_ptr)`
- `warp_Int warp_GetStatusDone (warp_Status status, warp_Int *done_ptr)`
- `warp_Int warp_GetNumIter (warp_Core core, warp_Int *niter_ptr)`
- `warp_Int warp_GetRNorm (warp_Core core, warp_Real *rnorm_ptr)`

8.2.1 Detailed Description

These are interface routines to initialize and run Warp

8.2.2 Typedef Documentation

8.2.2.1 `typedef struct _warp_Core_struct* warp_Core`

Points to the core structure defined in `_warp.h`

8.2.3 Function Documentation

8.2.3.1 warp_Int warp_Destroy (warp_Core *core*)

Clean up and destroy core.

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
-------------	-------------------------------

8.2.3.2 warp_Int warp_Drive(warp_Core *core*)

Carry out a simulation with Warp. Integrate in time.

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
-------------	-------------------------------

8.2.3.3 warp_Int warp_GetNumIter(warp_Core *core*, warp_Int * *niter_ptr*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>niter_ptr</i>	output, holds number of iterations taken

8.2.3.4 warp_Int warp_GetRNorm(warp_Core *core*, warp_Real * *rnorm_ptr*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>rnorm_ptr</i>	output, holds final residual norm

8.2.3.5 warp_Int warp_GetStatusDone(warp_Status *status*, warp_Int * *done_ptr*)

Return whether warp is done for the current status object

Parameters

<i>status</i>	structure containing current simulation info
<i>done_ptr</i>	output, =1 if warp has finished and this is the final Write, else =0

8.2.3.6 warp_Int warp_GetStatusIter(warp_Status *status*, warp_Int * *iter_ptr*)

Return the iteration for the current status object

Parameters

<i>status</i>	structure containing current simulation info
<i>iter_ptr</i>	output, current iteration number

8.2.3.7 warp_Int warp_GetStatusLevel(warp_Status *status*, warp_Int * *level_ptr*)

Return the warp level for the current status object

Parameters

<i>status</i>	structure containing current simulation info
<i>level_ptr</i>	output, current level in Warp

8.2.3.8 warp_Int warp_GetStatusResidual (warp_Status *status*, warp_Real * *rnorm_ptr*)

Return the residual for the current status object

Parameters

<i>status</i>	structure containing current simulation info
<i>rnorm_ptr</i>	output, current residual norm

8.2.3.9 warp_Int warp_Init (MPI_Comm *comm_world*, MPI_Comm *comm*, warp_Real *tstart*, warp_Real *tstop*, warp_Int *ntime*, warp_App *app*, warp_PtFcnPhi *phi*, warp_PtFcnInit *init*, warp_PtFcnClone *clone*, warp_PtFcnFree *free*, warp_PtFcnSum *sum*, warp_PtFcnDot *dot*, warp_PtFcnWrite *write*, warp_PtFcnBufSize *bufsize*, warp_PtFcnBufPack *bufpack*, warp_PtFcnBufUnpack *bufunpack*, warp_Core * *core_ptr*)

Create a core object with the required initial data.

The output is the *core_ptr* pointing to the newly created warp_Core structure.

Parameters

<i>comm_world</i>	Global communicator for space and time
<i>comm</i>	Communicator for temporal dimension
<i>tstart</i>	start time
<i>tstop</i>	End time
<i>ntime</i>	Initial number of temporal grid values
<i>app</i>	User-defined _warp_App structure
<i>phi</i>	User time stepping routine to advance state one time value
<i>init</i>	Initialize a warp_Vector function on finest temporal grid
<i>clone</i>	Clone a warp_Vector
<i>free</i>	Free a temporal state warp_Vector
<i>sum</i>	Compute vector sum of two temporal states
<i>dot</i>	Compute dot product between two temporal states
<i>write</i>	Writes (file, screen..) upon completion.
<i>bufsize</i>	Computes size for MPI buffer for one
<i>bufpack</i>	Packs MPI buffer to contain one temporal state
<i>bufunpack</i>	Unpacks MPI buffer containing one temporal state
<i>core_ptr</i>	Pointer to warp_Core (_warp_Core) struct

8.2.3.10 warp_Int warp_PrintStats (warp_Core *core*)

Print statistics after a Warp run.

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
-------------	-------------------------------

8.2.3.11 warp_Int warp_SetAbsTol (warp_Core *core*, warp_Real *atol*)

Set absolute stopping tolerance.

Recommended Option

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>atol</i>	absolute stopping tolerance

8.2.3.12 warp_Int warp_SetCFactor (warp_Core *core*, warp_Int *level*, warp_Int *cfactor*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>level</i>	level to set coarsening factor on
<i>cfactor</i>	desired coarsening factor

8.2.3.13 warp_Int warp_SetFMG (warp_Core *core*)

Use FMG cycling.

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
-------------	-------------------------------

8.2.3.14 warp_Int warp_SetLooseTol (warp_Core *core*, warp_Int *level*, warp_Real *loose_tol*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>level</i>	level to set loose_tol
<i>loose_tol</i>	tolerance to set

8.2.3.15 warp_Int warp_SetMaxCoarse (warp_Core *core*, warp_Int *max_coarse*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>max_coarse</i>	maximum coarse grid size

8.2.3.16 warp_Int warp_SetMaxIter (warp_Core *core*, warp_Int *max_iter*)

Set max number of multigrid iterations.

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>max_iter</i>	maximum iterations to allow

8.2.3.17 warp_Int warp_SetMaxLevels (warp_Core *core*, warp_Int *max_levels*)

Set max number of multigrid levels.

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>max_levels</i>	maximum levels

8.2.3.18 warp_Int warp_SetNFMGVcyc (warp_Core *core*, warp_Int *nfmg_Vcyc*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>nfmg_Vcyc</i>	number of V-cycles to do each FMG level

8.2.3.19 warp_Int warp_SetNRelax (warp_Core *core*, warp_Int *level*, warp_Int *nrelax*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>level</i>	level to set nrelax on
<i>nrelax</i>	number of relaxations to do on level

8.2.3.20 warp_Int warp_SetPrintFile (warp_Core *core*, const char * *printfile_name*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>printfile_name</i>	output file for Warp runtime output

8.2.3.21 warp_Int warp_SetPrintLevel (warp_Core *core*, warp_Int *print_level*)

Set print level for warp. This controls how much information is printed to standard out.

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

Default is level 1.

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>print_level</i>	desired print level

8.2.3.22 warp_Int warp_SetRelTol (warp_Core *core*, warp_Real *rtol*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>rtol</i>	relative stopping tolerance

8.2.3.23 warp_Int warp_SetSpatialCoarsen (warp_Core *core*, warp_PtFcnCoarsen *coarsen*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>coarsen</i>	function pointer to spatial coarsening routine

8.2.3.24 warp_Int warp_SetSpatialRefine (warp_Core *core*, warp_PtFcnRefine *refine*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>refine</i>	function pointer to spatial refinement routine

8.2.3.25 warp_Int warp_SetTightxTol (warp_Core *core*, warp_Int *level*, warp_Real *tight_tol*)

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

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>level</i>	level to set <i>tight_tol</i>
<i>tight_tol</i>	tolerance to set

8.2.3.26 warp_Int warp_SetWriteLevel (warp_Core *core*, warp_Int *write_level*)

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

Level 0: Never call the user's write routine Level 1: Only call the user's write routine after Warp is finished Level 2: Call the user's write routine every iteration in _warp_FRestrict(), which is during the down-cycle part of a Warp iteration

Default is level 1.

Parameters

<i>core</i>	warp_Core (_warp_Core) struct
<i>write_level</i>	desired write_level

8.2.3.27 warp_Int warp_SplitCommworld (const MPI_Comm * *comm_world*, warp_Int *px*, MPI_Comm * *comm_x*, MPI_Comm * *comm_t*)

Split MPI commworld into *comm_x* and *comm_t*, the spatial and temporal communicators

Parameters

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

8.3 Warp user test routines

Functions

- `warp_Int warp_TestInitWrite (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free)`
- `warp_Int warp_TestClone (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone)`
- `warp_Int warp_TestSum (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum)`
- `warp_Int warp_TestDot (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot)`
- `warp_Int warp_TestBuf (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnBufSize bufsize, warp_PtFcnBufPack bufpack, warp_PtFcnBufUnpack bufunpack)`
- `warp_Int warp_TestCoarsenRefine (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_Real fdt, warp_Real cdt, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnCoarsen coarsen, warp_PtFcnRefine refine)`
- `warp_Int warp_TestAll (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_Real fdt, warp_Real cdt, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnBufSize bufsize, warp_PtFcnBufPack bufpack, warp_PtFcnBufUnpack bufunpack, warp_PtFcnCoarsen coarsen, warp_PtFcnRefine refine)`

8.3.1 Detailed Description

These sanity check routines to help a user test their Warp code

8.3.2 Function Documentation

8.3.2.1 `warp_Int warp_TestAll (warp_App app, MPI_Comm comm_x, FILE * fp, warp_Real t, warp_Real fdt, warp_Real cdt, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnBufSize bufsize, warp_PtFcnBufPack bufpack, warp_PtFcnBufUnpack bufunpack, warp_PtFcnCoarsen coarsen, warp_PtFcnRefine refine)`

Runs all of the individual warp_Test* routines

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

Parameters

<code>app</code>	User defined App structure
<code>comm_x</code>	Spatial communicator
<code>fp</code>	File pointer (could be stdout or stderr) for log messages
<code>t</code>	Time value to initialize vector
<code>fdt</code>	Fine time step value that you spatially coarsen from
<code>cdt</code>	Coarse time step value that you coarsen to
<code>init</code>	Initialize a warp_Vector function on finest temporal grid
<code>free</code>	Free a temporal state warp_Vector

<i>clone</i>	Clone a temporal state warp_Vector
<i>sum</i>	Compute vector sum of two temporal states
<i>dot</i>	Compute dot product of two temporal states
<i>bufsize</i>	Computes size for MPI buffer for one
<i>bufpack</i>	Packs MPI buffer to contain one temporal state
<i>bufunpack</i>	Unpacks MPI buffer containing one temporal state
<i>coarsen</i>	Spatially coarsen a vector. If null, test is skipped.
<i>refine</i>	Spatially refine a vector. If null, test is skipped.

8.3.2.2 **warp_Int warp_TestBuf (warp_App app, MPI_Comm comm_x, FILE * fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnBufSize bufsize, warp_PtFcnBufPack bufpack, warp_PtFcnBufUnpack bufunpack)**

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

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

Parameters

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

8.3.2.3 **warp_Int warp_TestClone (warp_App app, MPI_Comm comm_x, FILE * fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone)**

Test Clone

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

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages
<i>t</i>	Time value to test clone with
<i>init</i>	Initialize a warp_Vector function on finest temporal grid
<i>write</i>	Write temporal state warp_Vector (can be NULL for no writing)
<i>free</i>	Free a temporal state warp_Vector
<i>clone</i>	Clone a temporal state warp_Vector

8.3.2.4 warp_Int warp_TestCoarsenRefine (warp_App app, MPI_Comm comm_x, FILE * fp, warp_Real t, warp_Real fdt, warp_Real cdt, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnCoarsen coarsen, warp_PtFcnRefine refine)

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

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

Parameters

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

8.3.2.5 warp_Int warp_TestDot (warp_App app, MPI_Comm comm_x, FILE * fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot)

Test Dot. A vector is initialized at time t and then cloned. Various dot products like $\langle 3 v, v \rangle / \langle v, v \rangle$ are computed with known output, e.g., $\langle 3 v, v \rangle / \langle v, v \rangle$ must equal 3. If all the tests pass, then 1 is returned.

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

Parameters

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

8.3.2.6 warp_Int warp_TestInitWrite (warp_App app, MPI_Comm comm_x, FILE * fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free)

Test Init, Write and Free

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

Parameters

<i>app</i>	User defined App structure
<i>comm_x</i>	Spatial communicator
<i>fp</i>	File pointer (could be stdout or stderr) for log messages
<i>t</i>	Time value to test init with
<i>init</i>	Initialize a warp_Vector function on finest temporal grid
<i>write</i>	Write temporal state warp_Vector (can be NULL for no writing)
<i>free</i>	Free a temporal state warp_Vector

8.3.2.7 **warp_Int warp_TestSum (warp_App app, MPI_Comm comm_x, FILE * fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum)**

Test Sum

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

Parameters

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

9 File Documentation

9.1 warp.h File Reference

Typedefs

- `typedef int warp_Int`
- `typedef double warp_Real`
- `typedef struct _warp_App_struct * warp_App`
- `typedef struct _warp_Vector_struct * warp_Vector`
- `typedef struct _warp_Status_struct * warp_Status`
- `typedef warp_Int(* warp_PtFcnPhi)(warp_App app, warp_Real tstart, warp_Real tstop, warp_Real accuracy, warp_Vector u, warp_Int *rfactor_ptr)`
- `typedef warp_Int(* warp_PtFcnInit)(warp_App app, warp_Real t, warp_Vector *u_ptr)`
- `typedef warp_Int(* warp_PtFcnClone)(warp_App app, warp_Vector u, warp_Vector *v_ptr)`
- `typedef warp_Int(* warp_PtFcnFree)(warp_App app, warp_Vector u)`
- `typedef warp_Int(* warp_PtFcnSum)(warp_App app, warp_Real alpha, warp_Vector x, warp_Real beta, warp_Vector y)`
- `typedef warp_Int(* warp_PtFcnDot)(warp_App app, warp_Vector u, warp_Vector v, warp_Real *dot_ptr)`
- `typedef warp_Int(* warp_PtFcnWrite)(warp_App app, warp_Real t, warp_Status status, warp_Vector u)`
- `typedef warp_Int(* warp_PtFcnBufSize)(warp_App app, warp_Int *size_ptr)`
- `typedef warp_Int(* warp_PtFcnBufPack)(warp_App app, warp_Vector u, void *buffer)`
- `typedef warp_Int(* warp_PtFcnBufUnpack)(warp_App app, void *buffer, warp_Vector *u_ptr)`
- `typedef warp_Int(* warp_PtFcnCoarsen)(warp_App app, warp_Real tstart, warp_Real f_tminus, warp_Real f_tplus, warp_Real c_tminus, warp_Real c_tplus, warp_Vector fu, warp_Vector *cu_ptr)`
- `typedef warp_Int(* warp_PtFcnRefine)(warp_App app, warp_Real tstart, warp_Real f_tminus, warp_Real f_tplus, warp_Real c_tminus, warp_Real c_tplus, warp_Vector cu, warp_Vector *fu_ptr)`
- `typedef struct _warp_Core_struct * warp_Core`

Functions

- `warp_Int warp_Init (MPI_Comm comm_world, MPI_Comm comm, warp_Real tstart, warp_Real tstop, warp_Int ntime, warp_App app, warp_PtFcnPhi phi, warp_PtFcnInit init, warp_PtFcnClone clone, warp_PtFcnFree free, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnWrite write, warp_PtFcnBufSize bufsize, warp_PtFcnBufPack bufpack, warp_PtFcnBufUnpack bufunpack, warp_Core *core_ptr)`
- `warp_Int warp_Drive (warp_Core core)`
- `warp_Int warp_Destroy (warp_Core core)`
- `warp_Int warp_PrintStats (warp_Core core)`
- `warp_Int warp_SetLoosexTol (warp_Core core, warp_Int level, warp_Real loose_tol)`
- `warp_Int warp_SetTightxTol (warp_Core core, warp_Int level, warp_Real tight_tol)`
- `warp_Int warp_SetMaxLevels (warp_Core core, warp_Int max_levels)`
- `warp_Int warp_SetMaxCoarse (warp_Core core, warp_Int max_coarse)`
- `warp_Int warp_SetAbsTol (warp_Core core, warp_Real atol)`
- `warp_Int warp_SetRelTol (warp_Core core, warp_Real rtol)`
- `warp_Int warp_SetNRelax (warp_Core core, warp_Int level, warp_Int nrelax)`
- `warp_Int warp_SetCFactor (warp_Core core, warp_Int level, warp_Int cfactor)`
- `warp_Int warp_SetMaxIter (warp_Core core, warp_Int max_iter)`
- `warp_Int warp_SetFMG (warp_Core core)`
- `warp_Int warp_SetNFMGVcyc (warp_Core core, warp_Int nfmvg_Vcyc)`

- `warp_Int warp_SetSpatialCoarsen (warp_Core core, warp_PtFcnCoarsen coarsen)`
- `warp_Int warp_SetSpatialRefine (warp_Core core, warp_PtFcnRefine refine)`
- `warp_Int warp_SetPrintLevel (warp_Core core, warp_Int print_level)`
- `warp_Int warp_SetPrintFile (warp_Core core, const char *printfile_name)`
- `warp_Int warp_SetWriteLevel (warp_Core core, warp_Int write_level)`
- `warp_Int warp_SplitCommworld (const MPI_Comm *comm_world, warp_Int px, MPI_Comm *comm_x, MPI_Comm *comm_t)`
- `warp_Int warp_GetStatusResidual (warp_Status status, warp_Real *rnorm_ptr)`
- `warp_Int warp_GetStatusIter (warp_Status status, warp_Int *iter_ptr)`
- `warp_Int warp_GetStatusLevel (warp_Status status, warp_Int *level_ptr)`
- `warp_Int warp_GetStatusDone (warp_Status status, warp_Int *done_ptr)`
- `warp_Int warp_GetNumIter (warp_Core core, warp_Int *niter_ptr)`
- `warp_Int warp_GetRNorm (warp_Core core, warp_Real *rnorm_ptr)`

9.1.1 Detailed Description

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

9.1.2 Typedef Documentation

9.1.2.1 `typedef int warp_Int`

Defines integer type

9.1.2.2 `typedef double warp_Real`

Defines floating point type

9.2 warp_test.h File Reference

Functions

- `warp_Int warp_TestInitWrite (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free)`
- `warp_Int warp_TestClone (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone)`
- `warp_Int warp_TestSum (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum)`
- `warp_Int warp_TestDot (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot)`
- `warp_Int warp_TestBuf (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnBufSize bufsize, warp_PtFcnBufPack bufpack, warp_PtFcnBufUnpack bufunpack)`
- `warp_Int warp_TestCoarsenRefine (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_Real fdt, warp_Real cdt, warp_PtFcnInit init, warp_PtFcnWrite write, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnCoarsen coarsen, warp_PtFcnRefine refine)`
- `warp_Int warp_TestAll (warp_App app, MPI_Comm comm_x, FILE *fp, warp_Real t, warp_Real fdt, warp_Real cdt, warp_PtFcnInit init, warp_PtFcnFree free, warp_PtFcnClone clone, warp_PtFcnSum sum, warp_PtFcnDot dot, warp_PtFcnBufSize bufsize, warp_PtFcnBufPack bufpack, warp_PtFcnBufUnpack bufunpack, warp_PtFcnCoarsen coarsen, warp_PtFcnRefine refine)`

Index

User interface routines, 17
 warp_Core, 17
 warp_Destroy, 18
 warp_Drive, 19
 warp_GetNumIter, 19
 warp_GetRNorm, 19
 warp_GetStatusDone, 19
 warp_GetStatusIter, 19
 warp_GetStatusLevel, 19
 warp_GetStatusResidual, 20
 warp_Init, 20
 warp_PrintStats, 20
 warp_SetAbsTol, 20
 warp_SetCFactor, 21
 warp_SetFMG, 21
 warp_SetLoosexTol, 21
 warp_SetMaxCoarse, 21
 warp_SetMaxIter, 21
 warp_SetMaxLevels, 21
 warp_SetNFMGVcyc, 22
 warp_SetNRelax, 22
 warp_SetPrintFile, 22
 warp_SetPrintLevel, 22
 warp_SetRelTol, 22
 warp_SetSpatialCoarsen, 23
 warp_SetSpatialRefine, 23
 warp_SetTightxTol, 23
 warp_SetWriteLevel, 23
 warp_SplitCommworld, 23

User-written routines, 14
 warp_App, 15
 warp_PtFcnBufPack, 15
 warp_PtFcnBufSize, 15
 warp_PtFcnBufUnpack, 15
 warp_PtFcnClone, 15
 warp_PtFcnCoarsen, 15
 warp_PtFcnDot, 15
 warp_PtFcnFree, 15
 warp_PtFcnInit, 15
 warp_PtFcnPhi, 15
 warp_PtFcnRefine, 15
 warp_PtFcnSum, 16
 warp_PtFcnWrite, 16
 warp_Status, 16
 warp_Vector, 16

Warp user test routines, 25
 warp_TestAll, 25
 warp_TestBuf, 26
 warp_TestClone, 26
 warp_TestCoarsenRefine, 26

 warp_TestDot, 27
 warp_TestInitWrite, 27
 warp_TestSum, 28

warp.h, 29
 warp_Int, 30
 warp_Real, 30

warp_App
 User-written routines, 15

warp_Core
 User interface routines, 17

warp_Destroy
 User interface routines, 18

warp_Drive
 User interface routines, 19

warp_GetNumIter
 User interface routines, 19

warp_GetRNorm
 User interface routines, 19

warp_GetStatusDone
 User interface routines, 19

warp_GetStatusIter
 User interface routines, 19

warp_GetStatusLevel
 User interface routines, 19

warp_Init
 User interface routines, 20

warp_Int
 warp.h, 30

warp_PrintStats
 User interface routines, 20

warp_PtFcnBufPack
 User-written routines, 15

warp_PtFcnBufSize
 User-written routines, 15

warp_PtFcnBufUnpack
 User-written routines, 15

warp_PtFcnClone
 User-written routines, 15

warp_PtFcnCoarsen
 User-written routines, 15

warp_PtFcnDot
 User-written routines, 15

warp_PtFcnFree
 User-written routines, 15

warp_PtFcnInit
 User-written routines, 15

warp_PtFcnPhi
 User-written routines, 15

warp_PtFcnRefine

User-written routines, 15
warp_PtFcnSum
 User-written routines, 16
warp_PtFcnWrite
 User-written routines, 16
warp_Real
 warp.h, 30
warp_SetAbsTol
 User interface routines, 20
warp_SetCFactor
 User interface routines, 21
warp_SetFMG
 User interface routines, 21
warp_SetLooseTol
 User interface routines, 21
warp_SetMaxCoarse
 User interface routines, 21
warp_SetMaxIter
 User interface routines, 21
warp_SetMaxLevels
 User interface routines, 21
warp_SetNFMGVcyc
 User interface routines, 22
warp_SetNRelax
 User interface routines, 22
warp_SetPrintFile
 User interface routines, 22
warp_SetPrintLevel
 User interface routines, 22
warp_SetRelTol
 User interface routines, 22
warp_SetSpatialCoarsen
 User interface routines, 23
warp_SetSpatialRefine
 User interface routines, 23
warp_SetTightxTol
 User interface routines, 23
warp_SetWriteLevel
 User interface routines, 23
warp_SplitCommworld
 User interface routines, 23
warp_Status
 User-written routines, 16
warp_TestAll
 Warp user test routines, 25
warp_TestBuf
 Warp user test routines, 26
warp_TestClone
 Warp user test routines, 26
warp_TestCoarsenRefine
 Warp user test routines, 26
warp_TestDot
 Warp user test routines, 27
warp_TestInitWrite