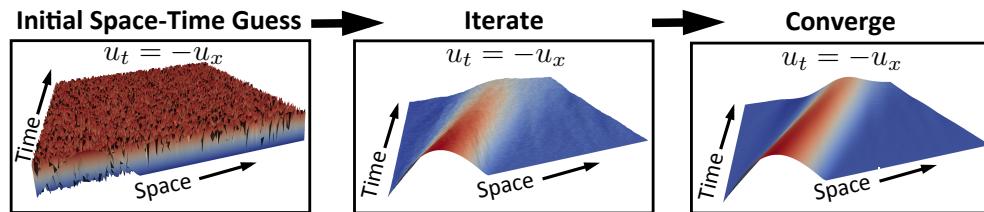


# Warp

Parallel-in-Time Software



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

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

Copyright (c) 2013, Lawrence Livermore National Security, LLC. Produced at the Lawrence Livermore National Laboratory. Written by the WARP team. All rights reserved.

This file is part of WARP. Please see the COPYRIGHT and LICENSE file for the copyright notice, disclaimer, contact information and the GNU Lesser General Public License.

WARP is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License (as published by the Free Software Foundation) version 2.1 dated February 1999.

WARP is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the IMPLIED WARRANTY OF MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the terms and conditions of the GNU General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

## Contents

<b>1 Abstract</b>	<b>2</b>
<b>2 Introduction</b>	<b>2</b>
2.1 Introduction . . . . .	2
2.2 Overview of Warp Algorithm . . . . .	2
2.2.1 Two-Grid Algorithm . . . . .	6
2.2.2 Summary . . . . .	6
2.3 Overview of Warp Code . . . . .	7
2.3.1 Parallel decomposition and memory . . . . .	7
2.3.2 Cycling and relaxation strategies . . . . .	8
2.3.3 Overlapping communication and computation . . . . .	9
2.3.4 Configuring the Warp Hierarchy . . . . .	9
2.3.5 Heat equation example . . . . .	10
2.4 Summary of Warp . . . . .	11
<b>3 Example</b>	<b>11</b>
<b>4 Building Warp</b>	<b>16</b>
<b>5 Compiling and running the examples</b>	<b>16</b>
<b>6 Coding Style</b>	<b>17</b>
<b>7 Using Doxygen</b>	<b>17</b>
<b>8 Regression Testing</b>	<b>18</b>
<b>9 Module Index</b>	<b>22</b>
9.1 Modules . . . . .	22
<b>10 Data Structure Index</b>	<b>22</b>
10.1 Data Structures . . . . .	22
<b>11 File Index</b>	<b>23</b>
11.1 File List . . . . .	23
<b>12 Module Documentation</b>	<b>23</b>
12.1 User-written routines . . . . .	23
12.1.1 Detailed Description . . . . .	23
12.1.2 Typedef Documentation . . . . .	24

12.2 User interface routines . . . . .	26
12.2.1 Detailed Description . . . . .	26
12.2.2 Typedef Documentation . . . . .	26
12.2.3 Function Documentation . . . . .	27
12.3 Warp test routines . . . . .	35
12.3.1 Detailed Description . . . . .	35
12.3.2 Function Documentation . . . . .	35
<b>13 Data Structure Documentation</b>	<b>39</b>
13.1 <code>_warp_AccuracyHandle</code> Struct Reference . . . . .	39
13.1.1 Detailed Description . . . . .	39
13.1.2 Field Documentation . . . . .	39
13.2 <code>_warp_CommHandle</code> Struct Reference . . . . .	39
13.2.1 Detailed Description . . . . .	40
13.2.2 Field Documentation . . . . .	40
13.3 <code>_warp_Core</code> Struct Reference . . . . .	40
13.3.1 Detailed Description . . . . .	41
13.3.2 Field Documentation . . . . .	41
13.4 <code>_warp_Grid</code> Struct Reference . . . . .	44
13.4.1 Detailed Description . . . . .	45
13.4.2 Field Documentation . . . . .	45
13.5 <code>_warp_Status</code> Struct Reference . . . . .	46
13.5.1 Detailed Description . . . . .	46
13.5.2 Field Documentation . . . . .	46
<b>14 File Documentation</b>	<b>47</b>
14.1 <code>_warp.h</code> File Reference . . . . .	47
14.1.1 Detailed Description . . . . .	48
14.1.2 Macro Definition Documentation . . . . .	48
14.1.3 Function Documentation . . . . .	49
14.1.4 Variable Documentation . . . . .	53
14.2 <code>util.h</code> File Reference . . . . .	53
14.2.1 Detailed Description . . . . .	53
14.2.2 Function Documentation . . . . .	53
14.3 <code>warp.h</code> File Reference . . . . .	54
14.3.1 Detailed Description . . . . .	55
14.3.2 Typedef Documentation . . . . .	55
14.4 <code>warp_test.h</code> File Reference . . . . .	55

14.4.1 Detailed Description . . . . .	56
---------------------------------------	----

**Index**

57

## 1 Abstract

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

It is **strongly recommended** that you also read [Parallel Time Integration with Multigrid](#) after reading the [Introduction](#). It is a more in depth discussion of the algorithm and associated experiments.

## 2 Introduction

### 2.1 Introduction

### 2.2 Overview of Warp Algorithm

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

To understand how 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 initial condition is a wave, and this wave propagates sequentially across space as time increases.

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

1. Relaxation on the fine grid, i.e., the grid that contains all of the desired time values
  - Relaxation is just a local application of the time stepping scheme, e.g., backward Euler
2. Restriction to the first coarse grid, i.e., interpolate the problem to a grid that contains fewer time values, say every second or every third time value
3. Relaxation on the first coarse grid

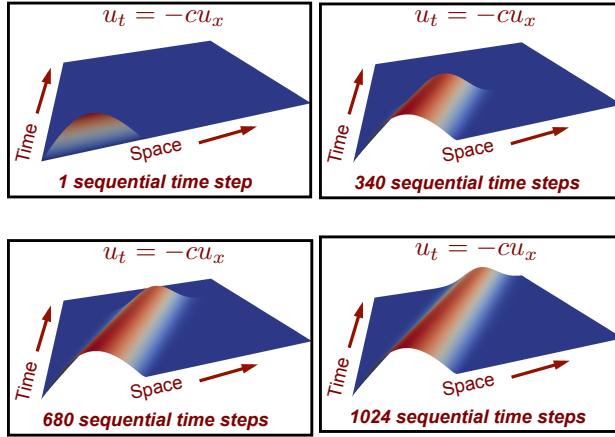


Figure 1: Sequential time stepping.

4. Restriction to the second coarse grid and so on...
5. When a coarse grid of trivial size (say 2 time steps) is reached, it is solved exactly.
6. The solution is then interpolated from the coarsest grid to the finest grid

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

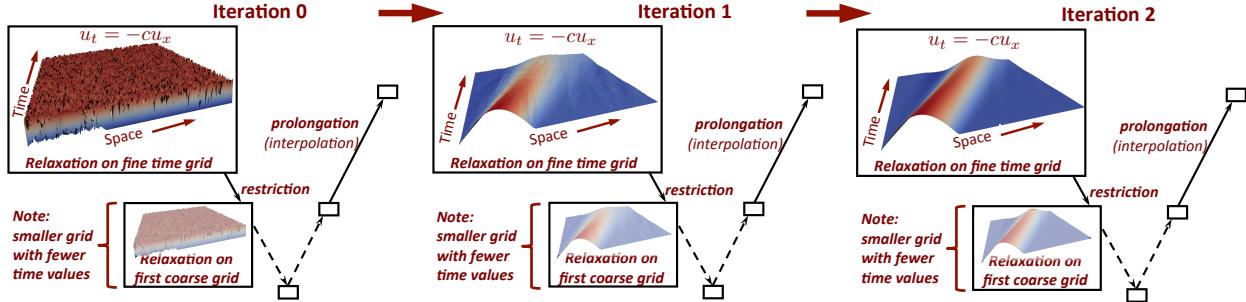


Figure 2: Warp iterations.

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. This is visible in the above figure by observing how the solution is updated from iteration 0 to iteration 1.
- Using coarser (cheaper) grids to correct the fine grid is 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, with evenly space time steps. 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

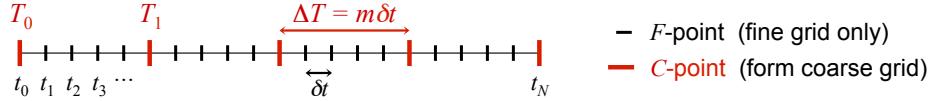
$$A\mathbf{u} = \mathbf{g}.$$

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

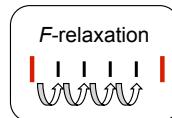
- nonintrusive, in that it coarsens only in time and the user defines  $\Phi$ 
  - Thus, users can continue using existing time stepping codes by wrapping them into our framework.
- optimal and  $O(N)$ , but  $O(N)$  with a higher constant than time stepping
  - Thus with enough computational resources, Warp will outperform sequential 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  $\Phi$  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. The first task is relaxation and an effective relaxation alternates between C and F sweeps (this is like line-



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

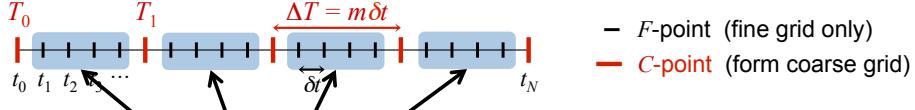


But, such an update can be done simultaneously over all F intervals in parallel, as depicted next.

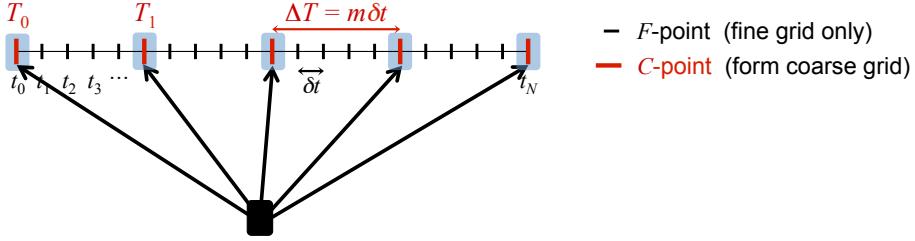
Following an F sweep we can also do C sweep, as depicted next.

---

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



- Update all F-points using time propagator  $\Phi$



- Update all C-points using time propagator  $\Phi$

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

- FCF or F-relaxation is highly parallel.
- But, a sequential component exists equaling 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, comes coarse grid correction. The restriction operator  $R$  maps fine grid quantities to the coarse grid by simply injecting values at C-points from the fine grid to the coarse grid,

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

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

The main question here is how to form the coarse grid matrix, which in turn asks how to define the coarse grid time stepper  $\Phi_\Delta$ . It is typical to let  $\Phi_\Delta$  simply be  $\Phi$  but with the coarse time step size  $\Delta T = m\delta t$ . Thus if

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

then

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

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

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

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

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

where  $m$  is the coarsening factor.

### 2.2.1 Two-Grid Algorithm

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

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

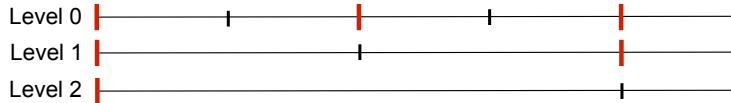
*Caveat:* The Warp implementation of FAS differs slightly from standard FAS. In standard FAS, the error is interpolated, using some standard interpolation formula like linear interpolation, to the fine points on the fine grid (here F-points). Instead, given our interpolation operator  $P_\Phi$ , we add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

### 2.2.2 Summary

In summary, a few points are

- Warp is an iterative solver for the global space-time problem.
- The user defines the time stepping routine  $\Phi$  and can wrap existing code to accomplish this.
- Warp convergence will depend heavily on how well  $\Phi_\Delta$  approximates  $\Phi^m$ , 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  $\delta t$ . This is a subject of research, but this approximation need not capture fine scale behavior, which is instead captured by relaxation on the fine grid.
- The coarsest grid is solved exactly, i.e., sequentially, which can be a bottleneck for two-level methods like Parareal,<sup>2</sup> 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 is ideal or exact, in that an application of interpolation leaves a zero residual at all F-points.
- The process is applied recursively until a trivially sized temporal grid is reached, e.g., 2 or 3 time points. Thus, the coarsening rate  $m$  determines how many levels there are in the hierarchy. For instance in this figure, a 3 level hierarchy is shown. Three levels are chosen because there are six time points,  $m = 2$  and  $m^2 < 6 \leq m^3$ . If the coarsening rate had been  $m = 4$  then there would only be two levels because, there would be no more points to coarsen!

- *F-point* (fine grid only)
- *C-point* (coarse grid)



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

## 2.3 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  $\Phi$  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.

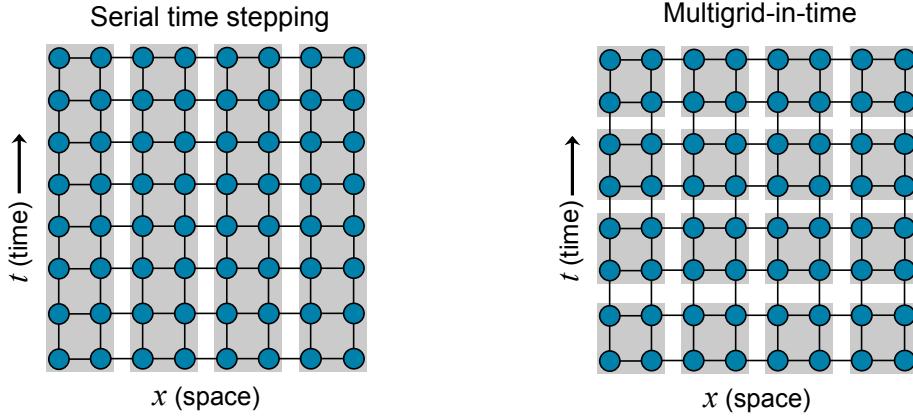
### Warp

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

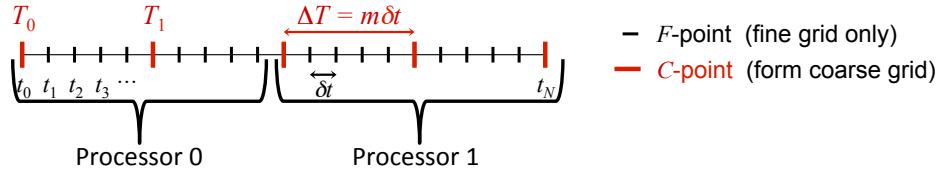
#### 2.3.1 Parallel decomposition and memory

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

<sup>2</sup> 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.



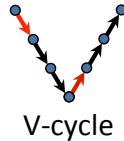
- Warp only handles temporal parallelism and is agnostic to the spatial decomposition. See [warp\\_SplitCommworld](#). Each processor owns a certain number of CF intervals of points, as depicted next, where each processor owns 2 CF intervals. Warp distributes Intervals evenly on the finest grid.



- Storage is greatly minimized by only storing C-points. Whenever an F-point is needed, it is generated by F-relaxation. That is, we only store the red C-point time values in the previous figure. Coarsening can 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.  
By only storing data at C-points, we effect a subtle change to the standard FAS algorithm (see [Two-Grid Algorithm](#)).
- In practice, storing only one space-time slab is advisable. That is, solve for as many time steps (say  $k$  time steps) as you have available memory for. Then move on to the next  $k$  time steps.

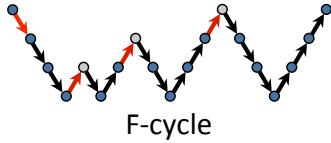
### 2.3.2 Cycling and relaxation strategies

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



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

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



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

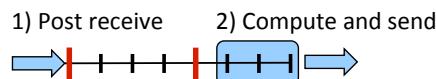
The number of FC relaxation sweeps is another important algorithmic setting. Note that at least one F-relaxation sweep is always done on a level. A few summary points about relaxation are as follows.

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

Last, [Parallel Time Integration with Multigrid](#) has a more in depth case study of cycling and relaxation strategies

### 2.3.3 Overlapping communication and computation

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



### 2.3.4 Configuring the Warp Hierarchy

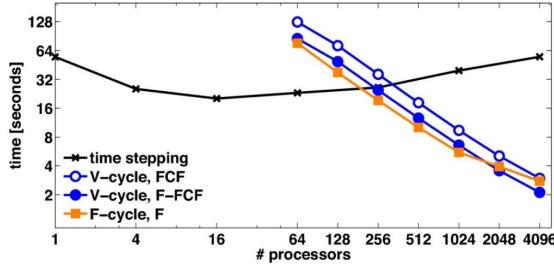
Some of the more basic Warp function calls allow you to control aspects discussed here.

- `warp_SetFMG`: switches between using F- and V-cycles.
- `warp_SetMaxIter`: sets the maximum number of Warp iterations
- `warp_SetCFactor`: sets the coarsening factor for any (or all levels)
- `warp_SetNRelax`: sets the number of CF-relaxation sweeps for any (or all levels)
- `warp_SetRelTol`, `warp_SetAbsTol`: sets the stopping tolerance

- `warp_SetMaxCoarse`: sets the maximum coarse grid size, in terms of C-points
- `warp_SetMaxLevels`: sets the maximum number of levels in the Warp hierarchy

### 2.3.5 Heat equation example

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



setup is as follows.

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

Regarding the performance, we can say

- The best speedup is 10x and this would grow if more processors were available.
- Although not shown, the iteration counts here are about 10-15 Warp iterations. See [Parallel Time Integration with Multigrid](#) for the exact iteration counts.
- At smaller core counts, serial time stepping is faster. But at about 256 processors, there is a crossover and Warp is faster.
- You can see the impact of the cycling and relaxation strategies discussed in [Cycling and relaxation strategies](#). For instance, even though *V-cycle, F-FCF* is a weaker relaxation strategy than *V-cycle, FCF* (i.e., the Warp convergence is slower), *V-cycle, F-FCF* has a faster time to solution than *V-cycle, FCF* because each cycle is cheaper.
- In general, one level of aggressive coarsening (here by a factor 16) followed by slower coarsening was found to be best on this machine.

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

## 2.4 Summary of Warp

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

## 3 Example

### A Simple Example

#### User Defined Structures and Wrappers

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

The two data structures are:

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

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

} my_App;
```

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

```
typedef struct _warp_Vector_struct
{
    double      value;

} my_Vector;
```

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

1. **Phi:** This function tells Warp how to take a time step, and is the core user routine. The user must advance the vector  $u$  from time  $tstart$  to time  $tstop$ . Here advancing the solution just involves the scalar  $\lambda$ . The *rfactor\_ptr* and *accuracy* parameters are advanced topics not used here.

**Importantly**, the  $g_i$  function (from [Overview of Warp Algorithm](#)) must be incorporated into Phi, so that  $\Phi(u_i) \rightarrow u_{i+1}$

```

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;
}

```

2. **Init:** This function tells Warp how to initialize 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;
}

```

3. **Clone:** This function tells Warp how to clone 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;
}

```

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

```

int
my_Free(warp_App      app,
        warp_Vector u)
{
    free(u);

    return 0;
}

```

5. **Sum:** This function tells Warp how to sum 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;
}
```

6. **Dot:** This function tells Warp how to take 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;
}
```

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

If `write_level` is 2 (see [warp\\_SetWriteLevel](#) ), then `Write` is called every Warp iteration and on every Warp level. In this case, `status` can be queried using the `warp_Get**Status()` functions, to determine the current Warp level and iteration. This allows for even more detailed tracking of the simulation.

See examples/drive-02 and examples/drive-04 for more advanced uses of the Write function. Drive-04 writes to a GLVIS visualization port, and examples/drive-02 writes to .vtu files.

```
int
my_Write(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, "%1.14e\n", (u->value));
    fflush(file);
    fclose(file);

    return 0;
}
```

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

```
int
my_BufSize(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;

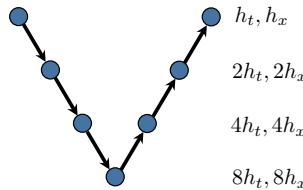
    u = (my_Vector *) malloc(sizeof(my_Vector));
    (u->value) = dbuffer[0];
    *u_ptr = u;

    return 0;
}

```

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

These functions allow you vary the spatial mesh size on Warp levels as depicted here where the spatial and temporal grid sizes are halved every level.



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

### Running Warp

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

```

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

```

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

```
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 are 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);
```

Finally, to run drive-01, type

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

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

### Testing Warp

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

At a lower level, to do sanity checks of your data structures and wrapper routines, there are also 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 and running the examples

Type

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

for instructions on how to run any driver.

To run the examples, type

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

1. drive-01 is the simplest example. It implements a scalar ODE and can be compiled and run with no outside dependencies.
2. drive-02 implements the 2D heat equation on a regular grid. You must have `hypre` installed and these variables in examples/Makefile set correctly  

```
HYPRE_DIR = ../../linear_solvers/hypre
HYPRE_FLAGS = -I$(HYPRE_DIR)/include
HYPRE_LIB = -L$(HYPRE_DIR)/lib -lHYPRE
```
3. drive-03 implements the 3D heat equation on a regular grid, and assumes `hypre` is installed just like drive-02.
4. drive-05 implements the 2D heat equation on a regular grid, but it uses spatial coarsening. This allows you to use explicit time stepping on each Warp level, regardless of time step size. It assumes `hypre` is installed just like drive-02.
5. drive-04 is a sophisticated test bed for various PDEs, mostly parabolic. It relies on the `mfem` package to create general finite element discretizations for the spatial problem. Other packages must be installed in this order.

- Unpack and install `Metis`
- Unpack and install `hypre`
- Unpack and install `mfem`. Make the serial version of mfem first by only typing `make`. Then make sure to set these variables correctly in the mfem Makefile:  

```
USE_METIS_5 = YES
HYPRE_DIR = where_ever_linear_solvers_is/hypre
```
- Make `GLVIS`, which needs serial mfem. Set these variables in the glvis makefile  

```
MFEM_DIR = mfem_location
MFEM_LIB = -L$(MFEM_DIR) -lmfem
```

- Go back to the mfem directory and type  

```
make clean
make parallel
```
- Go to warp/examples and set these Makefile variables,  

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

then type  

```
make drive-04
```
- To run drive-04 and glvis, open two windows. In one, start a glvis session  

```
./glvis
```

Then, in the other window, run drive-04  

```
mpirun -np ... drive-04 [args]
```

Glvis will listen on a port to which drive-04 will dump visualization information.

## 6 Coding Style

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

## 7 Using Doxygen

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

To make the documentation,

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

or to make a more extensive reference manual for developers,

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

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

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

or build doxygen from

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

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

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

### Documentation Strategy

- The doxygen comments are to be placed in the header files.

- A sample function declaration using the documentation approach using markdown (including typesetting equations) is in `warp.h` for the function `warp_Init()`
- A sample structure documentation is in `_warp.h` for `_warp_Core_struct`
- Descriptors for files can also be added, as at the top of `warp.h`
- The Doxygen manual is at <http://www.stack.nl/~dimitri/doxygen/manual/index.html>

### Warp Doxygen details

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

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

## 8 Regression Testing

### Overview

- There are three levels in the testing framework. At each level, the fine-grain output from a `testscript.sh` is dumped into a directory `testscript.dir`, with the standard out and error stored in `testscript.out` and `testscript.err`. The test `testscript.sh` passes if `testscript.err` is empty (nothing is written to standard error).
- Basic instructions: run a test with a command like  
`$ ./test.sh diffusion2D.sh`  
Then, see if `diffusion2D.err` is of size 0. If it is not, look at its contents to see which test failed.

- To add a new regression test, create a new lowest level script like `diffusion2D.sh` and then call it from a machine script at level 2.
- Regression tests should be run before pushing code. It is recommended to run the basic (lowest level) tests like `diffusion2d.sh` or machine test like `machine-tux.sh`

### Lowest Level Test Scripts

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

Files used:

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

Output:

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

At this level, we execute

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

or just

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

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

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

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

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

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

and the second `mpirun` test similarly writes the files

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

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

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

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

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

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

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

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

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

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

## Level 2 Scripts

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

Files used:

- `machine-tux.sh`

Output:

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

At this level, we execute

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

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

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

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

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

---

```
/usr/casc/hypre/warp/testing
```

### Level 3 Script

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

Files used:

- `autotest.sh`

Output:

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

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

There are four steps to running autotest.

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

### Cronfile

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

```
$ crontab -l
```

Next, append to `cronfile` whatever you already have

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

Finally, tell crontab to use your `cronfile`

```
$ crontab cronfile
```

Then make sure it took affect with

```
$ crontab -l
```

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

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

Jacob's crontab (on tux343):

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

## 9 Module Index

### 9.1 Modules

Here is a list of all modules:

<b>User-written routines</b>	<b>23</b>
<b>User interface routines</b>	<b>26</b>
<b>Warp test routines</b>	<b>35</b>

## 10 Data Structure Index

### 10.1 Data Structures

Here are the data structures with brief descriptions:

<b>_warp_AccuracyHandle</b>	<b>39</b>
<b>_warp_CommHandle</b>	<b>39</b>
<b>_warp_Core</b>	<b>40</b>
<b>_warp_Grid</b>	<b>44</b>
<b>_warp_Status</b>	<b>46</b>

## 11 File Index

### 11.1 File List

Here is a list of all files with brief descriptions:

<a href="#"><u>_warp.h</u></a>	Define headers for developer routines	47
<a href="#"><u>util.h</u></a>	Define headers for utility routines	53
<a href="#"><u>warp.h</u></a>	Define headers for user interface routines	54
<a href="#"><u>warp_test.h</u></a>	Define headers for Warp test routines	55

## 12 Module Documentation

### 12.1 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)`

#### 12.1.1 Detailed Description

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

### 12.1.2 Typedef Documentation

#### 12.1.2.1 `typedef struct _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.

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

This allows warp to send messages containing warp\_Vectors. This routine packs a vector  $u$  into a `void * buffer` for MPI.

#### 12.1.2.3 `typedef warp_Int(* warp_PtFcnBufSize)(warp_App app,warp_Int *size_ptr)`

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

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

This allows warp to receive messages containing warp\_Vectors. This routine unpacks a `void * buffer` from MPI into a warp\_Vector.

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

Clone  $u$  into  $v\_ptr$

#### 12.1.2.6 `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. The action of this function should match the `warp_PtFcnRefine` function.

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

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

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

Free and deallocate  $u$

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

Initializes a vector  $u\_ptr$  at time  $t$

#### 12.1.2.10 `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  $tstop$ . The `rfactor_ptr` and `accuracy` inputs are advanced topics. `rfactor_ptr` allows the user to tell Warp to refine this time interval.

---

12.1.2.11 `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. The action of this function should match the [warp\\_PtFcnCoarsen](#) function.

12.1.2.12 `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 -> y*

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

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

If *write\_level* is 2 (see [warp\\_SetWriteLevel](#) ), then *write* is called every Warp iteration and on every Warp level. In this case, *status* can be queried using the *warp\_Get\*\*Status()* functions, to determine the current warp level and iteration. This allows for even more detailed tracking of the simulation.

12.1.2.14 `typedef struct _warp_Status_struct* warp_Status`

Points to the status structure defined in [\\_warp.h](#) This is NOT a user-defined structure.

12.1.2.15 `typedef struct _warp_Vector_struct* warp_Vector`

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

## 12.2 User interface routines

### TypeDefs

- `typedef struct _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)`

### 12.2.1 Detailed Description

these are interface routines to initialize and run Warp

### 12.2.2 Typedef Documentation

#### 12.2.2.1 `typedef struct _warp_Core_struct* warp_Core`

points to the core structure defined in [\\_warp.h](#)

### 12.2.3 Function Documentation

#### 12.2.3.1 `warp_Int warp_Destroy ( warp_Core core )`

Clean up and destroy core.

## Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
-------------	---

**12.2.3.2 warp\_Int warp\_Drive ( warp\_Core *core* )**

Carry out a simulation with Warp. Integrate in time.

## Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
-------------	---

**12.2.3.3 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 ( <a href="#">_warp_Core</a> ) struct
<i>niter_ptr</i>	output, holds number of iterations taken

**12.2.3.4 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 ( <a href="#">_warp_Core</a> ) struct
<i>rnorm_ptr</i>	output, holds final residual norm

**12.2.3.5 warp\_Int warp\_GetStatusDone ( warp\_Status *status*, warp\_Int \* *done\_ptr* )**

Return whether warp is done for the current status object

*done\_ptr* = 1 indicates that this is the last call to Write, because Warp has stopped iterating (either maxiter has been reached, or the tolerance has been met).

## 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

**12.2.3.6 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

**12.2.3.7 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

**12.2.3.8 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

**12.2.3.9 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.

This core is used by Warp for internal data structures. The output is *core\_ptr* which points 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 a warp_Vector forward one step
<i>init</i>	Initialize a warp_Vector on the finest temporal grid
<i>clone</i>	Clone a warp_Vector
<i>free</i>	Free a warp_Vector
<i>sum</i>	Compute vector sum of two warp_Vectors
<i>dot</i>	Compute dot product between two warp_Vectors
<i>write</i>	Writes a warp_Vector to file, screen
<i>bufsize</i>	Computes size for MPI buffer for one warp_Vector
<i>bufpack</i>	Packs MPI buffer to contain one warp_Vector
<i>bufunpack</i>	Unpacks MPI buffer into a warp_Vector
<i>core_ptr</i>	Pointer to warp_Core (_warp_Core) struct

**12.2.3.10 warp\_Int warp\_PrintStats ( warp\_Core *core* )**

Print statistics after a Warp run.

## Parameters

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

**12.2.3.11 warp\_Int warp\_SetAbsTol ( warp\_Core *core*, warp\_Real *atol* )**

Set absolute stopping tolerance.

**Recommended option over relative tolerance**

## Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
<i>atol</i>	absolute stopping tolerance

12.2.3.12 warp\_Int warp\_SetCFactor ( warp\_Core *core*, warp\_Int *level*, warp\_Int *cfactor* )

Set the coarsening factor *cfactor* on grid *level* (level 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 ( <a href="#">_warp_Core</a> ) struct
<i>level</i>	<i>level</i> to set coarsening factor on
<i>cfactor</i>	desired coarsening factor

12.2.3.13 warp\_Int warp\_SetFMG ( warp\_Core *core* )

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

## Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
-------------	---

12.2.3.14 warp\_Int warp\_SetLooseTol ( warp\_Core *core*, warp\_Int *level*, warp\_Real *loose\_tol* )

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

## Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
<i>level</i>	<i>level</i> to set <i>loose_tol</i>
<i>loose_tol</i>	tolerance to set

12.2.3.15 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 ( <a href="#">_warp_Core</a> ) struct
<i>max_coarse</i>	maximum coarse grid size

12.2.3.16 warp\_Int warp\_SetMaxIter ( warp\_Core *core*, warp\_Int *max\_iter* )

Set max number of multigrid iterations.

## Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
<i>max_iter</i>	maximum iterations to allow

12.2.3.17 warp\_Int warp\_SetMaxLevels ( warp\_Core *core*, warp\_Int *max\_levels* )

Set max number of multigrid levels.

## Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
<i>max_levels</i>	maximum levels to allow

12.2.3.18 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 ( <a href="#">_warp_Core</a> ) struct
<i>nfmg_Vcyc</i>	number of V-cycles to do each FMG level

12.2.3.19 warp\_Int warp\_SetNRelax ( warp\_Core *core*, warp\_Int *level*, warp\_Int *nrelax* )

Set the number of relaxation sweeps *nrelax* on grid *level* (level 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 ( <a href="#">_warp_Core</a> ) struct
<i>level</i>	<i>level</i> to set <i>nrelax</i> on
<i>nrelax</i>	number of relaxations to do on <i>level</i>

12.2.3.20 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 ( <a href="#">_warp_Core</a> ) struct
<i>printfile_name</i>	output file for Warp runtime output

12.2.3.21 warp\_Int warp\_SetPrintLevel ( warp\_Core *core*, warp\_Int *print\_level* )

Set print level for warp. This controls how much information is printed to the Warp print file ([warp\\_SetPrintFile](#)).

- 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 ( <a href="#">_warp_Core</a> ) struct
<i>print_level</i>	desired print level

12.2.3.22 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 ( <a href="#">_warp_Core</a> ) struct
<i>rtol</i>	relative stopping tolerance

**12.2.3.23 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 ( <a href="#">_warp_Core</a> ) struct
<i>coarsen</i>	function pointer to spatial coarsening routine

**12.2.3.24 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 ( <a href="#">_warp_Core</a> ) struct
<i>refine</i>	function pointer to spatial refinement routine

**12.2.3.25 warp\_Int warp\_SetTightxTol ( warp\_Core *core*, warp\_Int *level*, warp\_Real *tight\_tol* )**

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

**Parameters**

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
<i>level</i>	level to set <i>tight_tol</i>
<i>tight_tol</i>	tolerance to set

**12.2.3.26 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 and on every level. This is during `_warp_FRestrict`, during the down-cycle part of a Warp iteration.

Default is level 1.

**Parameters**

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
<i>write_level</i>	desired write_level

**12.2.3.27 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. The total number of processors will equal Px\*Pt, where Px is the number of procs in space, and Pt is the number of procs in time.

**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)

## 12.3 Warp 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 )`

### 12.3.1 Detailed Description

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

### 12.3.2 Function Documentation

#### 12.3.2.1 `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 test vectors with

<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 on finest temporal grid
<i>free</i>	Free a warp_Vector
<i>clone</i>	Clone a warp_Vector
<i>sum</i>	Compute vector sum of two warp_Vectors
<i>dot</i>	Compute dot product of two warp_Vectors
<i>bufsize</i>	Computes size in bytes for one warp_Vector MPI buffer
<i>bufpack</i>	Packs MPI buffer to contain one warp_Vector
<i>bufunpack</i>	Unpacks MPI buffer into a warp_Vector
<i>coarsen</i>	Spatially coarsen a vector. If NULL, test is skipped.
<i>refine</i>	Spatially refine a vector. If NULL, test is skipped.

12.3.2.2 **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 )**

Test the BufPack, BufUnpack and BufSize functions.

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

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

#### Parameters

<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 on finest temporal grid
<i>free</i>	Free a warp_Vector
<i>sum</i>	Compute vector sum of two warp_Vectors
<i>dot</i>	Compute dot product of two warp_Vectors
<i>bufsize</i>	Computes size in bytes for one warp_Vector MPI buffer
<i>bufpack</i>	Packs MPI buffer to contain one warp_Vector
<i>bufunpack</i>	Unpacks MPI buffer containing one warp_Vector

12.3.2.3 **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 the clone function.

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

### Parameters

<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 (used to initialize the vectors)
<i>init</i>	Initialize a warp_Vector on finest temporal grid
<i>write</i>	Write a warp_Vector (can be NULL for no writing)
<i>free</i>	Free a warp_Vector
<i>clone</i>	Clone a warp_Vector

12.3.2.4 `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 )`

Test the Coarsen and Refine functions.

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

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

### Parameters

<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 test vectors
<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 on finest temporal grid
<i>write</i>	Write a warp_Vector (can be NULL for no writing)
<i>free</i>	Free a warp_Vector
<i>clone</i>	Clone a warp_Vector
<i>sum</i>	Compute vector sum of two warp_Vectors
<i>dot</i>	Compute dot product of two warp_Vectors
<i>coarsen</i>	Spatially coarsen a vector
<i>refine</i>	Spatially refine a vector

12.3.2.5 `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 the dot function.

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 on finest temporal grid
<i>free</i>	Free a warp_Vector
<i>clone</i>	Clone a warp_Vector
<i>sum</i>	Compute vector sum of two warp_Vectors
<i>dot</i>	Compute dot product of two warp_Vectors

12.3.2.6 `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 the init, write and free functions.

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 (used to initialize the vectors)
<i>init</i>	Initialize a warp_Vector on finest temporal grid
<i>write</i>	Write a warp_Vector (can be NULL for no writing)
<i>free</i>	Free a warp_Vector

12.3.2.7 `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 the sum function.

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

## Parameters

<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 on finest temporal grid
<i>write</i>	Write a warp_Vector (can be NULL for no writing)
<i>free</i>	Free a warp_Vector
<i>clone</i>	Clone a warp_Vector
<i>sum</i>	Compute vector sum of two warp_Vectors

## 13 Data Structure Documentation

### 13.1 `_warp_AccuracyHandle` Struct Reference

#### Data Fields

- `warp_Int` `matchF`
- `warp_Real` `value`
- `warp_Real` `old_value`
- `warp_Real` `loose`
- `warp_Real` `tight`
- `warp_Int` `tight_used`

#### 13.1.1 Detailed Description

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

#### 13.1.2 Field Documentation

##### 13.1.2.1 `warp_Real` `loose`

loose accuracy for spatial solves

##### 13.1.2.2 `warp_Int` `matchF`

##### 13.1.2.3 `warp_Real` `old_value`

old accuracy value used in FRestrict

##### 13.1.2.4 `warp_Real` `tight`

tight accuracy for spatial solves

##### 13.1.2.5 `warp_Int` `tight_used`

tight accuracy used (1) or not (0)

##### 13.1.2.6 `warp_Real` `value`

accuracy value

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

- `_warp.h`

### 13.2 `_warp_CommHandle` Struct Reference

#### Data Fields

- `warp_Int` `request_type`
- `warp_Int` `num_requests`

- MPI\_Request \* [requests](#)
- MPI\_Status \* [status](#)
- void \* [buffer](#)
- [warp\\_Vector](#) \* [vector\\_ptr](#)

### 13.2.1 Detailed Description

Warp comm handle structure

Used for initiating and completing nonblocking communication to pass warp\_Vectors between processors.

### 13.2.2 Field Documentation

#### 13.2.2.1 void\* buffer

Buffer for message

#### 13.2.2.2 warp\_Int num\_requests

number of active requests for this handle, usually 1

#### 13.2.2.3 warp\_Int request\_type

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

#### 13.2.2.4 MPI\_Request\* requests

MPI request structure

#### 13.2.2.5 MPI\_Status\* status

MPI status

#### 13.2.2.6 warp\_Vector\* vector\_ptr

warp\_vector being sent/received

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

- [\\_warp.h](#)

## 13.3 \_warp\_Core Struct Reference

### Data Fields

- 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_PtFcnCoarsen` coarsen
- `warp_PtFcnRefine` refine
- `warp_Int` write\_level
- `warp_Int` print\_level
- `warp_Int` max\_levels
- `warp_Int` max\_coarse
- `warp_Real` tol
- `warp_Int` rtol
- `warp_Int` \* nrels
- `warp_Int` nrdefault
- `warp_Int` \* cfactors
- `warp_Int` cfdefault
- `warp_Int` max\_iter
- `warp_Int` niter
- `warp_Real` rnorm
- `warp_Int` fmg
- `warp_Int` nfmg\_Vcyc
- `_warp_AccuracyHandle` \* accuracy
- `warp_Int` gupper
- `warp_Int` \* rfactors
- `warp_Int` nlevels
- `_warp_Grid` \*\* grids
- `warp_Real` localtime
- `warp_Real` globaltime

### 13.3.1 Detailed Description

The typedef `_warp_Core` struct is a **critical** part of warp and is passed to *each* routine in warp. It thus allows each routine access to Warp attributes.

### 13.3.2 Field Documentation

#### 13.3.2.1 `_warp_AccuracyHandle*` accuracy

accuracy of spatial solves on different levels

#### 13.3.2.2 `warp_App` app

application data for the user

#### 13.3.2.3 `warp_PtFcnBufPack` bufpack

pack a buffer

**13.3.2.4 `warp_PtFcnBufSize` bufsize**

return buffer size

**13.3.2.5 `warp_PtFcnBufUnpack` bufunpack**

unpack a buffer

**13.3.2.6 `warp_Int*` cfactors**

coarsening factors

**13.3.2.7 `warp_Int` cfdefault**

default coarsening factor

**13.3.2.8 `warp_PtFcnClone` clone**

clone a vector

**13.3.2.9 `warp_PtFcnCoarsen` coarsen**

(optional) return a coarsened vector

**13.3.2.10 `MPI_Comm` comm**

communicator for the time dimension

**13.3.2.11 `MPI_Comm` comm\_world****13.3.2.12 `warp_PtFcnDot` dot**

dot product

**13.3.2.13 `warp_Int` fmg**

use FMG cycle

**13.3.2.14 `warp_PtFcnFree` free**

free up a vector

**13.3.2.15 `warp_Real` globaltime**

global wall time for [warp\\_Drive\(\)](#)

**13.3.2.16 `_warp_Grid**` grids**

pointer to temporal grid structures for each level

**13.3.2.17 `warp_Int` gupper**

global upper index on the fine grid

**13.3.2.18 `warp_PtFcnInit` init**

return an initialized warp\_Vector

**13.3.2.19 warp\_Real localtime**

local wall time for [warp\\_Drive\(\)](#)

**13.3.2.20 warp\_Int max\_coarse**

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

**13.3.2.21 warp\_Int max\_iter**

maximum number of multigrid in time iterations

**13.3.2.22 warp\_Int max\_levels**

maximum number of temporal grid levels

**13.3.2.23 warp\_Int nfmg\_Vcyc**

number of V-cycle calls at each level in FMG

**13.3.2.24 warp\_Int niter**

number of iterations

**13.3.2.25 warp\_Int nlevels**

number of temporal grid levels

**13.3.2.26 warp\_Int nrdefault**

default number of pre-relaxations

**13.3.2.27 warp\_Int\* nrels**

number of pre-relaxations on each level

**13.3.2.28 warp\_Int ntime**

initial number of time intervals

**13.3.2.29 warp\_PtFcnPhi phi**

apply phi function

**13.3.2.30 warp\_Int print\_level**

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

**13.3.2.31 warp\_PtFcnRefine refine**

(optional) return a refined vector

**13.3.2.32 warp\_Int\* rfactors**

refinement factors for finest grid (if any)

13.3.2.33 `warp_Real rnorm`

residual norm

13.3.2.34 `warp_Int rtol`

use relative tolerance

13.3.2.35 `warp_PtFcnSum sum`

vector sum

13.3.2.36 `warp_Real tol`

stopping tolerance

13.3.2.37 `warp_Real tstart`

start time

13.3.2.38 `warp_Real tstop`

stop time

13.3.2.39 `warp_PtFcnWrite write`

write the vector

13.3.2.40 `warp_Int write_level`

determines how often to call the user's write routine

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

- [\\_warp.h](#)

## 13.4 `_warp_Grid` Struct Reference

### Data Fields

- `warp_Int level`
- `warp_Int ilower`
- `warp_Int iupper`
- `warp_Int clower`
- `warp_Int cupper`
- `warp_Int cfactor`
- `warp_Int ncpoints`
- `warp_Vector * ua`
- `warp_Real * ta`
- `warp_Vector * va`
- `warp_Vector * wa`
- `warp_Int recv_index`
- `warp_Int send_index`
- `_warp_CommHandle * recv_handle`
- `_warp_CommHandle * send_handle`

- `warp_Vector * ua_alloc`
- `warp_Real * ta_alloc`
- `warp_Vector * va_alloc`
- `warp_Vector * wa_alloc`

### 13.4.1 Detailed Description

Warp Grid structure for a certain time level

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

### 13.4.2 Field Documentation

#### 13.4.2.1 `warp_Int cfactor`

coarsening factor

#### 13.4.2.2 `warp_Int clower`

smallest C point index

#### 13.4.2.3 `warp_Int cupper`

largest C point index

#### 13.4.2.4 `warp_Int ilower`

smallest time index at this level

#### 13.4.2.5 `warp_Int iupper`

largest time index at this level

#### 13.4.2.6 `warp_Int level`

Level that grid is on

#### 13.4.2.7 `warp_Int ncpoints`

number of C points

#### 13.4.2.8 `_warp_CommHandle* recv_handle`

Handle for nonblocking receives of warp\_Vectors

#### 13.4.2.9 `warp_Int recv_index`

-1 means no receive

#### 13.4.2.10 `_warp_CommHandle* send_handle`

Handle for nonblocking sends of warp\_Vectors

#### 13.4.2.11 `warp_Int send_index`

-1 means no send

**13.4.2.12 `warp_Real*` ta**

time values (all points)

**13.4.2.13 `warp_Real*` ta\_alloc**

original memory allocation for ta

**13.4.2.14 `warp_Vector*` ua**

unknown vectors (C-points only)

**13.4.2.15 `warp_Vector*` ua\_alloc**

original memory allocation for ua

**13.4.2.16 `warp_Vector*` va**

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

**13.4.2.17 `warp_Vector*` va\_alloc**

original memory allocation for va

**13.4.2.18 `warp_Vector*` wa**

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

**13.4.2.19 `warp_Vector*` wa\_alloc**

original memory allocation for wa

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

- [\\_warp.h](#)

## 13.5 `_warp_Status` Struct Reference

### Data Fields

- `warp_Int iter`
- `warp_Int level`
- `warp_Real rnorm`
- `warp_Int done`

### 13.5.1 Detailed Description

Points to the status structure which defines the status of Warp at a given instant on a some level during a run. The user accesses it through `warp_Get**Status()` functions.

### 13.5.2 Field Documentation

#### 13.5.2.1 `warp_Int done`

boolean describing whether warp has finished

### 13.5.2.2 `warp_Int iter`

warp iteration number

### 13.5.2.3 `warp_Int level`

current level in warp

### 13.5.2.4 `warp_Real rnorm`

residual norm

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

- [\\_warp.h](#)

## 14 File Documentation

### 14.1 `_warp.h` File Reference

#### Data Structures

- [struct \\_warp\\_Status](#)
- [struct \\_warp\\_CommHandle](#)
- [struct \\_warp\\_AccuracyHandle](#)
- [struct \\_warp\\_Grid](#)
- [struct \\_warp\\_Core](#)

#### Macros

- [#define \\_warp\\_CommHandleElt\(handle, elt\) \(\(handle\) -> elt\)](#)
- [#define \\_warp\\_GridElt\(grid, elt\) \(\(grid\) -> elt\)](#)
- [#define \\_warp\\_StatusElt\(status, elt\) \( \(status\) -> elt \)](#)
- [#define \\_warp\\_CoreElt\(core, elt\) \( \(core\) -> elt \)](#)
- [#define \\_warp\\_CoreFcn\(core, fcn\) \(\\*\(\(core\) -> fcn\)\)](#)
- [#define \\_warp\\_TAlloc\(type, count\) \( \(type \\*\)malloc\(\(size\\_t\)\(sizeof\(type\) \\* \(count\)\)\) \)](#)
- [#define \\_warp\\_CTAalloc\(type, count\) \( \(type \\*\)calloc\(\(size\\_t\)\(count\), \(size\\_t\)sizeof\(type\)\) \)](#)
- [#define \\_warp\\_TReAlloc\(ptr, type, count\) \( \(type \\*\)realloc\(\(char \\*\)ptr, \(size\\_t\)\(sizeof\(type\) \\* \(count\)\)\) \)](#)
- [#define \\_warp\\_TFree\(ptr\) \( free\(\(char \\*\)ptr\), ptr = NULL \)](#)
- [#define \\_warp\\_MapFineToCoarse\(findex, cfactor, cindex\) \( cindex = \(findex\)/\(cfactor\) \)](#)
- [#define \\_warp\\_MapCoarseToFine\(cindex, cfactor, findex\) \( findex = \(cindex\)\\*\(cfactor\) \)](#)
- [#define \\_warp\\_IsFPoint\(index, cfactor\) \( \(index\)% \(cfactor\) \)](#)
- [#define \\_warp\\_IsCPoint\(index, cfactor\) \( !\\_warp\\_IsFPoint\(index, cfactor\) \)](#)

#### Functions

- [warp\\_Int \\_warp\\_GetDistribution \(warp\\_Core core, warp\\_Int \\*ilower\\_ptr, warp\\_Int \\*iupper\\_ptr\)](#)
- [warp\\_Int \\_warp\\_GetProc \(warp\\_Core core, warp\\_Int level, warp\\_Int index, warp\\_Int \\*proc\\_ptr\)](#)
- [warp\\_Int \\_warp\\_CommRecvInit \(warp\\_Core core, warp\\_Int level, warp\\_Int index, warp\\_Vector \\*vector\\_ptr, \\_warp\\_CommHandle \\*\\*handle\\_ptr\)](#)

- `warp_Int _warp_CommSendInit (warp_Core core, warp_Int level, warp_Int index, warp_Vector vector, _warp_CommHandle **handle_ptr)`
- `warp_Int _warp_CommWait (warp_Core core, _warp_CommHandle **handle_ptr)`
- `warp_Int _warp_UCommInit (warp_Core core, warp_Int level)`
- `warp_Int _warp_UCommInitF (warp_Core core, warp_Int level)`
- `warp_Int _warp_UCommWait (warp_Core core, warp_Int level)`
- `warp_Int _warp_UGetInterval (warp_Core core, warp_Int level, warp_Int interval_index, warp_Int *flo_ptr, warp_Int *fhi_ptr, warp_Int *ci_ptr)`
- `warp_Int _warp_UGetVectorRef (warp_Core core, warp_Int level, warp_Int index, warp_Vector *u_ptr)`
- `warp_Int _warp_USetVectorRef (warp_Core core, warp_Int level, warp_Int index, warp_Vector u)`
- `warp_Int _warp_UGetVector (warp_Core core, warp_Int level, warp_Int index, warp_Vector *u_ptr)`
- `warp_Int _warp_USetVector (warp_Core core, warp_Int level, warp_Int index, warp_Vector u)`
- `warp_Int _warp_UWriteVector (warp_Core core, warp_Int level, warp_Int index, warp_Status status, warp_Vector u)`
- `warp_Int _warp_Phi (warp_Core core, warp_Int level, warp_Int index, warp_Real accuracy, warp_Vector u, warp_Int *rfactor)`
- `warp_Int _warp_Step (warp_Core core, warp_Int level, warp_Int index, warp_Real accuracy, warp_Vector u)`
- `warp_Int _warp_Coarsen (warp_Core core, warp_Int level, warp_Int f_index, warp_Int c_index, warp_Vector fvector, warp_Vector *cvector)`
- `warp_Int _warp_Refine (warp_Core core, warp_Int level, warp_Int f_index, warp_Int c_index, warp_Vector cvector, warp_Vector *fvector)`
- `warp_Int _warp_GridInit (warp_Core core, warp_Int level, warp_Int ilower, warp_Int iupper, _warp_Grid **grid_ptr)`
- `warp_Int _warp_GridDestroy (warp_Core core, _warp_Grid *grid)`
- `warp_Int _warp_InitGuess (warp_Core core, warp_Int level)`
- `warp_Int _warp_CFRelax (warp_Core core, warp_Int level)`
- `warp_Int _warp_FRestrict (warp_Core core, warp_Int level, warp_Int iter, warp_Real *rnorm_ptr)`
- `warp_Int _warp_FInterp (warp_Core core, warp_Int level, warp_Int iter, warp_Real rnorm)`
- `warp_Int _warp_FRefine (warp_Core core, warp_Int *refined_ptr)`
- `warp_Int _warp_FWrite (warp_Core core, warp_Real rnorm, warp_Int iter, warp_Int level, warp_Int done)`
- `warp_Int _warp_InitHierarchy (warp_Core core, _warp_Grid *fine_grid)`
- `warp_Int _warp_InitStatus (warp_Real rnorm, warp_Int iter, warp_Int level, warp_Int done, warp_Status *status_ptr)`
- `warp_Int _warp_DestroyStatus (warp_Status status)`

## Variables

- `warp_Int _warp_error_flag`
- `FILE * _warp_printfile`

### 14.1.1 Detailed Description

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

### 14.1.2 Macro Definition Documentation

#### 14.1.2.1 `#define _warp_CommHandleElt( handle, elt ) ((handle) -> elt)`

Accessor for `_warp_CommHandle` attributes

14.1.2.2 `#define _warp_CoreElt( core, elt ) ((core) -> elt)`

Accessor for `_warp_Core` attributes

14.1.2.3 `#define _warp_CoreFcn( core, fcn ) (*((core) -> fcn))`

Accessor for `_warp_Core` functions

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

Allocation macro

14.1.2.5 `#define _warp_GridElt( grid, elt ) ((grid) -> elt)`

Accessor for `_warp_Grid` attributes

14.1.2.6 `#define _warp_IsCPoint( index, cfactor ) (!_warp_IsFPoint(index, cfactor))`

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

14.1.2.7 `#define _warp_IsFPoint( index, cfactor ) ((index)%(cfactor) == 0)`

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

14.1.2.8 `#define _warp_MapCoarseToFine( cindex, cfactor, findex ) ( findex = (cindex)*(cfactor) )`

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

14.1.2.9 `#define _warp_MapFineToCoarse( finDEX, cfactor, cindex ) ( cindex = (finDEX)/(cfactor) )`

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

14.1.2.10 `#define _warp_StatusElt( status, elt ) ((status) -> elt)`

Accessor for `_warp_Status` attributes

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

Allocation macro

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

Free memory macro

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

Re-allocation macro

### 14.1.3 Function Documentation

14.1.3.1 `warp_Int warp_CFRelax ( warp_Core core, warp_Int level )`

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

### 14.1.3.2 `warp_Int_warp_Coarsen ( warp_Core core, warp_Int level, warp_Int f_index, warp_Int c_index, warp_Vector fvector, warp_Vector * cvector )`

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

### 14.1.3.3 `warp_Int_warp_CommRecvInit ( warp_Core core, warp_Int level, warp_Int index, warp_Vector * vector_ptr, _warp_CommHandle ** handle_ptr )`

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

### 14.1.3.4 `warp_Int_warp_CommSendInit ( warp_Core core, warp_Int level, warp_Int index, warp_Vector vector, _warp_CommHandle ** handle_ptr )`

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

### 14.1.3.5 `warp_Int_warp_CommWait ( warp_Core core, _warp_CommHandle ** handle_ptr )`

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

### 14.1.3.6 `warp_Int_warp_DestroyStatus ( warp_Status status )`

Destroy a warp\_Status structure

### 14.1.3.7 `warp_Int_warp_Finterp ( warp_Core core, warp_Int level, warp_Int iter, warp_Real rnorm )`

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

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

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

#### Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
<i>level</i>	interp from level to level+1
<i>iter</i>	current iteration number (for user info)
<i>rnorm</i>	residual norm (if level 0)

### 14.1.3.8 `warp_Int_warp_FRefine ( warp_Core core, warp_Int * refined_ptr )`

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

RDF: Todo, routine is unwritten

### 14.1.3.9 `warp_Int_warp_FRestrict ( warp_Core core, warp_Int level, warp_Int iter, warp_Real * rnorm_ptr )`

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

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

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

If *level==0*, then *rnorm\_ptr* will contain the residual norm.

### Parameters

<i>core</i>	warp_Core ( <a href="#">_warp_Core</a> ) struct
<i>level</i>	restrict from level to level+1
<i>iter</i>	current iteration number (for user info)
<i>rnorm_ptr</i>	pointer to residual norm (if level 0)

14.1.3.10 `warp_Int_warp_FWrite( warp_Core core, warp_Real rnorm, warp_Int iter, warp_Int level, warp_Int done )`

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

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

14.1.3.11 `warp_Int_warp_GetDistribution( warp_Core core, warp_Int * ilower_ptr, warp_Int * iupper_ptr )`

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

14.1.3.12 `warp_Int_warp_GetProc( warp_Core core, warp_Int level, warp_Int index, warp_Int * proc_ptr )`

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

14.1.3.13 `warp_Int_warp_GridDestroy( warp_Core core, _warp_Grid * grid )`

Destroy a Warp *grid*

14.1.3.14 `warp_Int_warp_GridInit( warp_Core core, warp_Int level, warp_Int ilower, warp_Int iupper, _warp_Grid ** grid_ptr )`

Create a new grid object *grid\_ptr* in *core* at *level*

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

14.1.3.15 `warp_Int_warp_InitGuess( warp_Core core, warp_Int level )`

Set initial guess at C-points on *level*

14.1.3.16 `warp_Int_warp_InitHierarchy( warp_Core core, _warp_Grid * fine_grid )`

Initialize (and re-initialize) hierarchy

14.1.3.17 `warp_Int_warp_InitStatus( warp_Real rnorm, warp_Int iter, warp_Int level, warp_Int done, warp_Status * status_ptr )`

Initialize a warp\_Status structure in *status\_ptr*, setting the status values *rnorm*, *iter*, *level*, *done*

14.1.3.18 `warp_Int_warp_Phi( warp_Core core, warp_Int level, warp_Int index, warp_Real accuracy, warp_Vector u, warp_Int * rfactor )`

Apply Phi to the vector *u*

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

---

14.1.3.19 `warp_Int_warp_Refine( warp_Core core, warp_Int level, warp_Int f_index, warp_Int c_index, warp_Vector cvector, warp_Vector *fvector )`

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

14.1.3.20 `warp_Int_warp_Step( warp_Core core, warp_Int level, warp_Int index, warp_Real accuracy, warp_Vector u )`

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

14.1.3.21 `warp_Int_warp_UCommInit( warp_Core core, warp_Int level )`

Working on all intervals

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

14.1.3.22 `warp_Int_warp_UCommInitF( warp_Core core, warp_Int level )`

Working only on F-pt intervals

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

14.1.3.23 `warp_Int_warp_UCommWait( warp_Core core, warp_Int level )`

Finish up communication

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

14.1.3.24 `warp_Int_warp_UGetInterval( warp_Core core, warp_Int level, warp_Int interval_index, warp_Int *flo_ptr, warp_Int *fhi_ptr, warp_Int *ci_ptr )`

Retrieve the time step indices at this *level* which correspond to the FC interval given by *interval\_index*. *ci\_ptr* is the time step index for the C point and *flo\_ptr* and *fhi\_ptr* are the smallest and largest F point indices in this interval. *flo = ci + 1*, and *fhi = ci + coarsening\_factor - 1*

14.1.3.25 `warp_Int_warp_UGetVector( warp_Core core, warp_Int level, warp_Int index, warp_Vector *u_ptr )`

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

14.1.3.26 `warp_Int_warp_UGetVectorRef( warp_Core core, warp_Int level, warp_Int index, warp_Vector *u_ptr )`

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

14.1.3.27 `warp_Int_warp_USetVector( warp_Core core, warp_Int level, warp_Int index, warp_Vector u )`

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

### 14.1.3.28 `warp_Int_warp_USetVectorRef ( warp_Core core, warp_Int level, warp_Int index, warp_Vector u )`

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

### 14.1.3.29 `warp_Int_warp_UWriteVector ( warp_Core core, warp_Int level, warp_Int index, warp_Status status, warp_Vector u )`

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

## 14.1.4 Variable Documentation

### 14.1.4.1 `warp_Int_warp_error_flag`

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

### 14.1.4.2 `FILE* _warp_printfile`

This is the print file for redirecting std::cout for all Warp screen output

## 14.2 util.h File Reference

### Functions

- `warp_Int_warp_ProjectInterval (warp_Int ilower, warp_Int iupper, warp_Int index, warp_Int stride, warp_Int *pilower, warp_Int *piupper)`
- `warp_Int_warp_SetAccuracy (warp_Real rnorm, warp_Real loose_tol, warp_Real tight_tol, warp_Real old_Accuracy, warp_Real tol, warp_Real *paccuracy)`
- `warp_Int_warp_printf (const char *format, ...)`
- `warp_Int_warp_ParPrintfFlush (FILE *file, warp_Int myid, char *message, ...)`

### 14.2.1 Detailed Description

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

### 14.2.2 Function Documentation

#### 14.2.2.1 `warp_Int_warp_ParPrintfFlush ( FILE * file, warp_Int myid, char * message, ... )`

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

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

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

#### 14.2.2.2 `warp_Int_warp_printf ( const char * format, ... )`

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

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

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

#### 14.2.2.3 `warp_Int_warp_ProjectInterval ( warp_Int ilower, warp_Int iupper, warp_Int index, warp_Int stride, warp_Int * pilower, warp_Int * piupper )`

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

#### 14.2.2.4 `warp_Int_warp_SetAccuracy ( warp_Real rnorm, warp_Real loose_tol, warp_Real tight_tol, warp_Real oldAccuracy, warp_Real tol, warp_Real * paccuracy )`

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

## 14.3 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)`

#### 14.3.1 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.

#### 14.3.2 Typedef Documentation

##### 14.3.2.1 `typedef int warp_Int`

Defines integer type

##### 14.3.2.2 `typedef double warp_Real`

Defines floating point type

### 14.4 `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)`

#### 14.4.1 Detailed Description

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

## Index

\_warp.h, 47  
    \_warp\_CFRelax, 49  
    \_warp\_CTAalloc, 49  
    \_warp\_Coarsen, 49  
    \_warp\_CommHandleElt, 48  
    \_warp\_CommRecvInit, 50  
    \_warp\_CommSendInit, 50  
    \_warp\_CommWait, 50  
    \_warp\_CoreElt, 48  
    \_warp\_CoreFcn, 49  
    \_warp\_DestroyStatus, 50  
    \_warp\_FInterp, 50  
    \_warp\_FRefine, 50  
    \_warp\_FRestrict, 50  
    \_warp\_FWrite, 51  
    \_warp\_GetDistribution, 51  
    \_warp\_GetProc, 51  
    \_warp\_GridDestroy, 51  
    \_warp\_GridElt, 49  
    \_warp\_GridInit, 51  
    \_warp\_InitGuess, 51  
    \_warp\_InitHierarchy, 51  
    \_warp\_InitStatus, 51  
    \_warp\_IsCPoint, 49  
    \_warp\_IsFPoint, 49  
    \_warp\_MapCoarseToFine, 49  
    \_warp\_MapFineToCoarse, 49  
    \_warp\_Phi, 51  
    \_warp\_Refine, 51  
    \_warp\_StatusElt, 49  
    \_warp\_Step, 52  
    \_warp\_TAlloc, 49  
    \_warp\_TFree, 49  
    \_warp\_TReAlloc, 49  
    \_warp\_UCommInit, 52  
    \_warp\_UCommInitF, 52  
    \_warp\_UCommWait, 52  
    \_warp\_UGetInterval, 52  
    \_warp\_UGetVector, 52  
    \_warp\_UGetVectorRef, 52  
    \_warp\_USetVector, 52  
    \_warp\_USetVectorRef, 52  
    \_warp\_UWriteVector, 53  
    \_warp\_error\_flag, 53  
    \_warp\_printfile, 53  
  
\_warp\_AccuracyHandle, 39  
    loose, 39  
    matchF, 39  
    old\_value, 39  
    tight, 39  
    tight\_used, 39  
  
                value, 39  
\_warp\_CFRelax  
    \_warp.h, 49  
\_warp\_CTAalloc  
    \_warp.h, 49  
\_warp\_Coarsen  
    \_warp.h, 49  
\_warp\_CommHandle, 39  
    buffer, 40  
    num\_requests, 40  
    request\_type, 40  
    requests, 40  
    status, 40  
    vector\_ptr, 40  
\_warp\_CommHandleElt  
    \_warp.h, 48  
\_warp\_CommRecvInit  
    \_warp.h, 50  
\_warp\_CommSendInit  
    \_warp.h, 50  
\_warp\_CommWait  
    \_warp.h, 50  
\_warp\_Core, 40  
    accuracy, 41  
    app, 41  
    bufpack, 41  
    bufsize, 41  
    bufunpack, 42  
    cfactors, 42  
    cfdefault, 42  
    clone, 42  
    coarsen, 42  
    comm, 42  
    comm\_world, 42  
    dot, 42  
    fmg, 42  
    free, 42  
    globaltime, 42  
    grids, 42  
    gupper, 42  
    init, 42  
    localtime, 42  
    max\_coarse, 43  
    max\_iter, 43  
    max\_levels, 43  
    nfmf\_Vcyc, 43  
    niter, 43  
    nlevels, 43  
    nrdefault, 43  
    nrels, 43  
    ntime, 43

phi, 43  
print\_level, 43  
refine, 43  
rfactors, 43  
rnorm, 43  
rtol, 44  
sum, 44  
tol, 44  
tstart, 44  
tstop, 44  
write, 44  
write\_level, 44  
`_warp_CoreElt`  
    `_warp.h`, 48  
`_warp_CoreFcn`  
    `_warp.h`, 49  
`_warp_DestroyStatus`  
    `_warp.h`, 50  
`_warp_FInterp`  
    `_warp.h`, 50  
`_warp_FRefine`  
    `_warp.h`, 50  
`_warp_FRestrict`  
    `_warp.h`, 50  
`_warp_FWrite`  
    `_warp.h`, 51  
`_warp_GetDistribution`  
    `_warp.h`, 51  
`_warp_GetProc`  
    `_warp.h`, 51  
`_warp_Grid`, 44  
    cfactor, 45  
    clower, 45  
    cupper, 45  
    ilower, 45  
    iupper, 45  
    level, 45  
    ncpoints, 45  
    recv\_handle, 45  
    recv\_index, 45  
    send\_handle, 45  
    send\_index, 45  
    ta, 45  
    ta\_alloc, 46  
    ua, 46  
    ua\_alloc, 46  
    va, 46  
    va\_alloc, 46  
    wa, 46  
    wa\_alloc, 46  
`_warp_GridDestroy`  
    `_warp.h`, 51  
`_warp_GridElt`  
    `_warp.h`, 49  
`_warp_GridInit`  
    `_warp.h`, 51  
`_warp_InitGuess`  
    `_warp.h`, 51  
`_warp_InitHierarchy`  
    `_warp.h`, 51  
`_warp_InitStatus`  
    `_warp.h`, 51  
`_warp_IsCPoint`  
    `_warp.h`, 49  
`_warp_IsFPoint`  
    `_warp.h`, 49  
`_warp_MapCoarseToFine`  
    `_warp.h`, 49  
`_warp_MapFineToCoarse`  
    `_warp.h`, 49  
`_warp_ParPrintfFlush`  
    `util.h`, 53  
`_warp_Phi`  
    `_warp.h`, 51  
`_warp_ProjectInterval`  
    `util.h`, 54  
`_warp_Refine`  
    `_warp.h`, 51  
`_warp_SetAccuracy`  
    `util.h`, 54  
`_warp_Status`, 46  
    done, 46  
    iter, 46  
    level, 47  
    rnorm, 47  
`_warp_StatusElt`  
    `_warp.h`, 49  
`_warp_Step`  
    `_warp.h`, 52  
`_warp_TAlloc`  
    `_warp.h`, 49  
`_warp_TFree`  
    `_warp.h`, 49  
`_warp_TReAlloc`  
    `_warp.h`, 49  
`_warp_UCommInit`  
    `_warp.h`, 52  
`_warp_UCommInitF`  
    `_warp.h`, 52  
`_warp_UCommWait`  
    `_warp.h`, 52  
`_warp_UGetInterval`  
    `_warp.h`, 52  
`_warp_UGetVector`  
    `_warp.h`, 52  
`_warp_UGetVectorRef`  
    `_warp.h`, 52  
`_warp_USetVector`

\_warp.h, 52  
 \_warp\_USetVectorRef  
     \_warp.h, 52  
 \_warp\_UWriteVector  
     \_warp.h, 53  
 \_warp\_error\_flag  
     \_warp.h, 53  
 \_warp\_printf  
     util.h, 53  
 \_warp\_printfile  
     \_warp.h, 53

accuracy  
     \_warp\_Core, 41  
 app  
     \_warp\_Core, 41

buffer  
     \_warp\_CommHandle, 40  
 bufpack  
     \_warp\_Core, 41  
 bufsize  
     \_warp\_Core, 41  
 bufunpack  
     \_warp\_Core, 42

cfactor  
     \_warp\_Grid, 45  
 cfactors  
     \_warp\_Core, 42  
 cfdefault  
     \_warp\_Core, 42

clone  
     \_warp\_Core, 42  
 clower  
     \_warp\_Grid, 45  
 coarsen  
     \_warp\_Core, 42

comm  
     \_warp\_Core, 42  
 comm\_world  
     \_warp\_Core, 42

cupper  
     \_warp\_Grid, 45

done  
     \_warp\_Status, 46  
 dot  
     \_warp\_Core, 42

fmg  
     \_warp\_Core, 42  
 free  
     \_warp\_Core, 42

globaltime  
     \_warp\_Core, 42  
 grids  
     \_warp\_Core, 42  
 gupper  
     \_warp\_Core, 42

ilower  
     \_warp\_Grid, 45  
 init  
     \_warp\_Core, 42  
 iter  
     \_warp\_Status, 46

iupper  
     \_warp\_Grid, 45

level  
     \_warp\_Grid, 45  
     \_warp\_Status, 47  
 localtime  
     \_warp\_Core, 42  
 loose  
     \_warp\_AccuracyHandle, 39

matchF  
     \_warp\_AccuracyHandle, 39

max\_coarse  
     \_warp\_Core, 43  
 max\_iter  
     \_warp\_Core, 43  
 max\_levels  
     \_warp\_Core, 43

ncpoints  
     \_warp\_Grid, 45  
 nfmg\_Vcyc  
     \_warp\_Core, 43  
 niter  
     \_warp\_Core, 43  
 nlevels  
     \_warp\_Core, 43  
 nrdefault  
     \_warp\_Core, 43  
 nrels  
     \_warp\_Core, 43  
 ntime  
     \_warp\_Core, 43  
 num\_requests  
     \_warp\_CommHandle, 40

old\_value  
     \_warp\_AccuracyHandle, 39

phi  
     \_warp\_Core, 43

print\_level  
  \_warp\_Core, 43

recv\_handle  
  \_warp\_Grid, 45

recv\_index  
  \_warp\_Grid, 45

refine  
  \_warp\_Core, 43

request\_type  
  \_warp\_CommHandle, 40

requests  
  \_warp\_CommHandle, 40

rfactors  
  \_warp\_Core, 43

rnorm  
  \_warp\_Core, 43  
  \_warp\_Status, 47

rtol  
  \_warp\_Core, 44

send\_handle  
  \_warp\_Grid, 45

send\_index  
  \_warp\_Grid, 45

status  
  \_warp\_CommHandle, 40

sum  
  \_warp\_Core, 44

ta  
  \_warp\_Grid, 45

ta\_alloc  
  \_warp\_Grid, 46

tight  
  \_warp\_AccuracyHandle, 39

tight\_used  
  \_warp\_AccuracyHandle, 39

tol  
  \_warp\_Core, 44

tstart  
  \_warp\_Core, 44

tstop  
  \_warp\_Core, 44

ua  
  \_warp\_Grid, 46

ua\_alloc  
  \_warp\_Grid, 46

User interface routines, 26  
  warp\_Core, 26  
  warp\_Destroy, 27  
  warp\_Drive, 28  
  warp\_GetNumIter, 28  
  warp\_GetRNorm, 28

warp\_GetStatusDone, 28  
warp\_GetStatusIter, 28  
warp\_GetStatusLevel, 28  
warp\_GetStatusResidual, 29  
warp\_Init, 29  
warp\_PrintStats, 29  
warp\_SetAbsTol, 29  
warp\_SetCFactor, 31  
warp\_SetFMG, 31  
warp\_SetLoosexTol, 31  
warp\_SetMaxCoarse, 31  
warp\_SetMaxIter, 31  
warp\_SetMaxLevels, 31  
warp\_SetNFMGVcyc, 32  
warp\_SetNRelax, 32  
warp\_SetPrintFile, 32  
warp\_SetPrintLevel, 32  
warp\_SetRelTol, 32  
warp\_SetSpatialCoarsen, 33  
warp\_SetSpatialRefine, 33  
warp\_SetTightxTol, 33  
warp\_SetWriteLevel, 33  
warp\_SplitCommworld, 33

User-written routines, 23  
  warp\_App, 24  
  warp\_PtFcnBufPack, 24  
  warp\_PtFcnBufSize, 24  
  warp\_PtFcnBufUnpack, 24  
  warp\_PtFcnClone, 24  
  warp\_PtFcnCoarsen, 24  
  warp\_PtFcnDot, 24  
  warp\_PtFcnFree, 24  
  warp\_PtFcniInit, 24  
  warp\_PtFcniPhi, 24  
  warp\_PtFcniRefine, 24  
  warp\_PtFcniSum, 25  
  warp\_PtFcniWrite, 25  
  warp\_Status, 25  
  warp\_Vector, 25

util.h, 53  
  \_warp\_ParPrintfFlush, 53  
  \_warp\_ProjectInterval, 54  
  \_warp\_SetAccuracy, 54  
  \_warp\_printf, 53

va  
  \_warp\_Grid, 46

va\_alloc  
  \_warp\_Grid, 46

value  
  \_warp\_AccuracyHandle, 39

vector\_ptr  
  \_warp\_CommHandle, 40

wa

\_warp\_Grid, 46  
**wa\_alloc**  
 \_warp\_Grid, 46  
**Warp test routines**, 35  
 warp\_TestAll, 35  
 warp\_TestBuf, 36  
 warp\_TestClone, 36  
 warp\_TestCoarsenRefine, 37  
 warp\_TestDot, 37  
 warp\_TestInitWrite, 38  
 warp\_TestSum, 38  
**warp.h**, 54  
 warp\_Int, 55  
 warp\_Real, 55  
**warp\_App**  
 User-written routines, 24  
**warp\_Core**  
 User interface routines, 26  
**warp\_Destroy**  
 User interface routines, 27  
**warp\_Drive**  
 User interface routines, 28  
**warp\_GetNumIter**  
 User interface routines, 28  
**warp\_GetRNorm**  
 User interface routines, 28  
**warp\_GetStatusDone**  
 User interface routines, 28  
**warp\_GetStatusIter**  
 User interface routines, 28  
**warp\_GetStatusLevel**  
 User interface routines, 28  
**warp\_GetStatusResidual**  
 User interface routines, 29  
**warp\_Init**  
 User interface routines, 29  
**warp\_Int**  
 warp.h, 55  
**warp\_PrintStats**  
 User interface routines, 29  
**warp\_PtFcnBufPack**  
 User-written routines, 24  
**warp\_PtFcnBufSize**  
 User-written routines, 24  
**warp\_PtFcnBufUnpack**  
 User-written routines, 24  
**warp\_PtFcnClone**  
 User-written routines, 24  
**warp\_PtFcnCoarsen**  
 User-written routines, 24  
**warp\_PtFcnDot**  
 User-written routines, 24  
**warp\_PtFcnFree**  
 User-written routines, 24  
**warp\_PtFcnInit**  
 User-written routines, 24  
**warp\_PtFcnPhi**  
 User-written routines, 24  
**warp\_PtFcnRefine**  
 User-written routines, 24  
**warp\_PtFcnSum**  
 User-written routines, 25  
**warp\_PtFcnWrite**  
 User-written routines, 25  
**warp\_Real**  
 warp.h, 55  
**warp\_SetAbsTol**  
 User interface routines, 29  
**warp\_SetCFactor**  
 User interface routines, 31  
**warp\_SetFMG**  
 User interface routines, 31  
**warp\_SetLooseTol**  
 User interface routines, 31  
**warp\_SetMaxCoarse**  
 User interface routines, 31  
**warp\_SetMaxIter**  
 User interface routines, 31  
**warp\_SetMaxLevels**  
 User interface routines, 31  
**warp\_SetNFMGVcyc**  
 User interface routines, 32  
**warp\_SetNRelax**  
 User interface routines, 32  
**warp\_SetPrintFile**  
 User interface routines, 32  
**warp\_SetPrintLevel**  
 User interface routines, 32  
**warp\_SetRelTol**  
 User interface routines, 32  
**warp\_SetSpatialCoarsen**  
 User interface routines, 33  
**warp\_SetSpatialRefine**  
 User interface routines, 33  
**warp\_SetTightxTol**  
 User interface routines, 33  
**warp\_SetWriteLevel**  
 User interface routines, 33  
**warp\_SplitCommworld**  
 User interface routines, 33  
**warp\_Status**  
 User-written routines, 25  
**warp\_TestAll**  
 Warp test routines, 35  
**warp\_TestBuf**  
 Warp test routines, 36  
**warp\_TestClone**  
 Warp test routines, 36

warp\_TestCoarsenRefine  
    Warp test routines, 37  
warp\_TestDot  
    Warp test routines, 37  
warp\_TestInitWrite  
    Warp test routines, 38  
warp\_TestSum  
    Warp test routines, 38  
warp\_Vector  
    User-written routines, 25  
warp\_test.h, 55  
write  
    \_warp\_Core, 44  
write\_level  
    \_warp\_Core, 44