1.摘要

对于具有进化行为的离散化的线性问题，这个软件包采用最佳缩放的多重网格解法。通常，演化方程的解决方法基于时间游走方法，就是通过一个个时间步迭代求解方程。在传统的时间积分上并行受限于空间的并行程度。然而，在计算机体系中目前的趋势是研究拥有更多处理器的系统，但是不一定能运算的更快。因此，想要更短的总体运行时间需要更佳的并行程度。多重网格就是实现在时间上并行的一种方法，但拓展经典的多重网格方法只对椭圆算子有意义。在这个软件中，我们通过减少网格实现一种非入侵并且最佳缩放的时间并行方法。软件包中的例子说明，用MGRIT(multigrid-reduction-in-time)方法解决各种二维三维方程是最佳的。在这些例子中我们也能看到，相比于现代体系中的顺序时间游走，MGRIT 能实现更好的加速。

2.3 XBraid 算法概述

XBraid 的目标是解决一个问题比传统的时间游走算法更快。与顺序时间游走算法不同，在迭代的同时，XBraid 在所有时间节点值上更新一个时空的猜想解来解决问题。这个初始的猜想解可以是

任何值，甚至是时空里的一个随机函数。通过构造时间网格的等级制度来迭代更新猜想解，而细网格包含了所有模拟过程的时间节点。每一个后续的网格是一个包含更少时间节点的粗网格。粗网格包含的时间节点数量更少，并且求解方程更快。在粗网格上时间游走问题的解能被用来修正初始的细网格上的解。与空间的多重网格方法相似，粗网格的修正只起修正和加快细网格解的收敛的速度。在粗网格的右边，不必要展示出一个精确的时间离散格式。因此，一个拥有许多(上千或更多)时间节点的问题能在 10 到 15 次 XBraid 迭代后得到解，并且求解的时间能大幅度减少。然而，需要消耗大量计算资源来加速求解。

为了理解 XBraid 与传统的时间游走方法的不同，考虑线性平流方程 $u_t = -cu_x$，下一幅图中描述了用顺序时间点迭代得到的解的图像。初始条件是一个波，并且随着时间的增加，该波将在整个空间中依次传播。

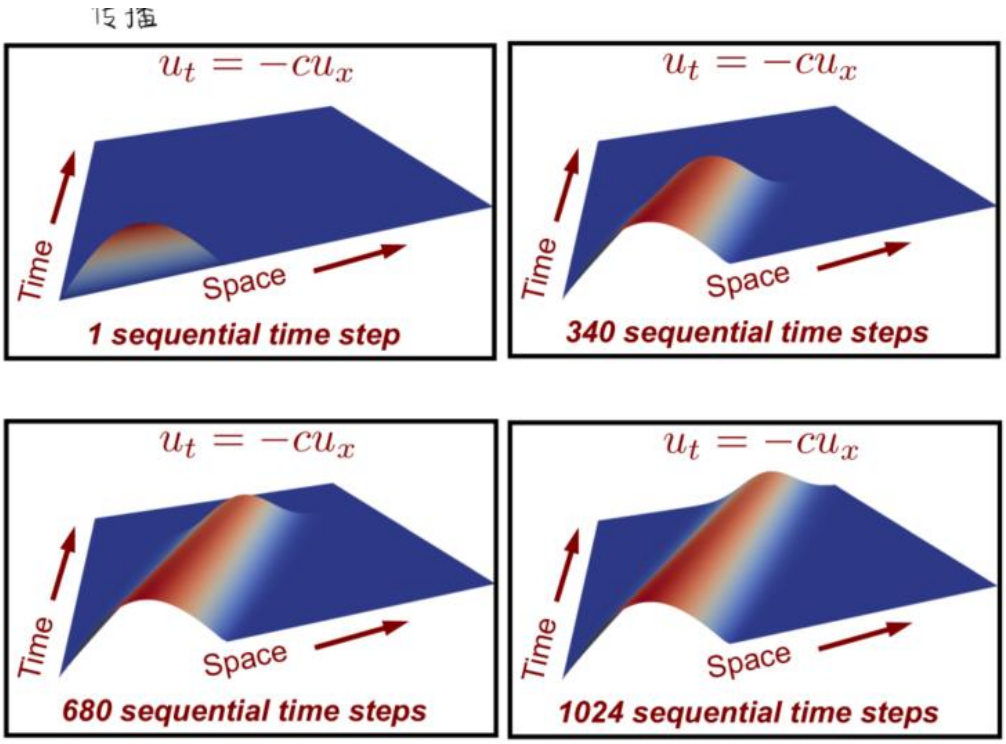


Figure 1 Sequential time stepping.

相反，XBraid 从整个时空的猜想解开始，为便于演示，我们将其设为随机。XBraid 迭代执行

1. 在细网格上松弛，即包含所有期望时间值的网格。松弛只是时间步进方案的局部应用，例如向后欧拉。
2. 对第一个粗网格限制，即将问题插到包含较少时间节点的网格，例如每两个或每三个时间节点。
3. 在第一个粗网格上松弛
4. 在第二个粗网格限制，如此进行下去

5. 当达到平凡大小的粗网格（比如说 2 个时间步长）时，问题就完全被解决了。

6. 问题的解就是从最粗的网格插值到最细的网格。

一次 XBraid 迭代称为一次循环，这些循环继续直到解足够准确为止。在下图中可看出，对于这个简单问题仅需要几次迭代。

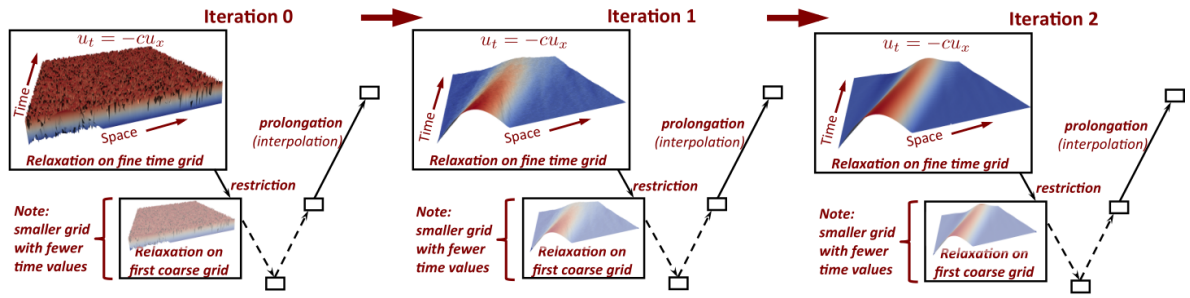


Figure 2 XBraid iterations.

有几个要点可以做

•粗时间网格只需一次 XBraid 迭代，就能跨时空进行全局信息传播。可以在上图中看到，求解方法如何从 Iteration0 更新到 Iteration1。

•使用粗(cheaper)网格修正细网格类似于空间的多重网格方法。

• 时间节点个数超过 1024 个时，很少步 XBraid 迭代就能发现解。因此，如果有足够的处理器可用于并行化 XBraid，我们可以看到在传统时间步长上的加速（稍后会详细介绍）。

•上述只是一个简单的例子，时间节点剖分是均匀的。而 XBraid 的结构是可以处理可变时间步长和自适应时间步长。

假设你有一个普通的常微分方程，

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

令 $t_i = \mathrm{i}\delta t,\quad i = 0,1,\ldots,N$ 是一个时间网格，步长 $\delta t = T/N$，并且 u_i 大约估计值为 $u(t_i)$，一个普通的一步时间离散法为：

$$u_0 = g_0$$

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

传统的时间游走方法第一步算出 $i=1$，然后算 $i=2$，如此下去。对于线性时间算子 $\{\Phi_i\}$ 这也可以直接表示为直接求解下述系统：

$$Au = \begin{pmatrix} I & & & \\ -\Phi_1 & I & & \\ & \ddots & \ddots & \\ & & -\Phi_N & I \end{pmatrix} \begin{pmatrix} u_o \\ u_1 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g_o \\ g_1 \\ \vdots \\ g_N \end{pmatrix} \equiv g$$

或者

$$Au = g$$

这个过程是最优的并且是O(N)的，但是是顺序的。XBraid通过一个最优多重网格约简迭代法，来代替顺序求解，实现在时间上的并行。

这个方法有如下特点：

•非侵入式，因为算法只粗化时间节点和用户定义的Φ。因此，用户可以通过将现有的时间步进代码包装到我们的框架中，来继续使用他们。

• 最佳的并且是O(N)的，但是O(N)是比时间迭代更高的常数。因此在具有足够的计算资源时，XBraid 将胜过顺序时间游走。

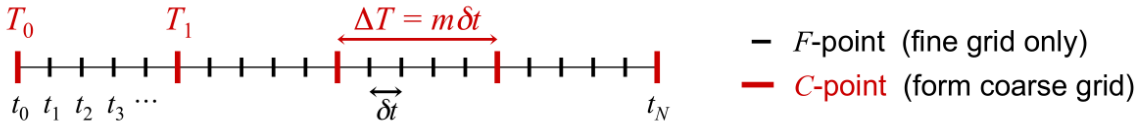
• 高度并行

现在我们更详细的描述两层网格过程，多级模拟是该过程的递归应用。我们也假设Φ是不变的或者符号简单的。XBraid以粗化因子m>1在时间维度粗化，来产生一个具有$N_\Delta = N/m$ 个点的时间网格，并且时间步长$\Delta T = m\delta t$，相应的粗网格问题，

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

是通过定义粗网格算子$\{\Phi_\Delta\}$而得到的，粗网格算子至少与细网格算子$\{\Phi\}$一样简单。与矩阵 A 相比，A_Δ 具有更少的行和列，例如，如果每隔两个时间节点粗化网格，A_Δ 只有一半的行和列。

这个粗时间网格引起细网格划分为 C 点（与粗网格点相关联）和 F 点，下图是可视化。C 点在粗网格和细网格中都存在，而 F 点只在细网格上。



每一个多重网格算法都要求有松弛方法和在网格中传递数值的方法。我们的松弛方案在所谓的 F-松弛和 C-松弛之间交替，具体如下所示。F-松弛更新在区间(T_i, T_{i+1}) F 点的值$\{u_j\}$，仅通过时间传播算子$\{\Phi\}$ 简单的在区间内传播 C 点的值u_{mi}。但是这个一个顺序迭代过程，每个 F 点间隔的更新彼此独立，并且可以并行计算。相似的，C-松弛基于 F 点的值u_{mi-1}更新 C 点的值u_{mi}，并且这种更新也可以并行计算。可以将这种松弛方法视为空间中的线性松弛，因为对于整个时间步长，残差都设置为 0。F 点更新同时并行完成，

如下所示。进行 F 点扫描后，C 点更新也同时并行完成，如下所示。

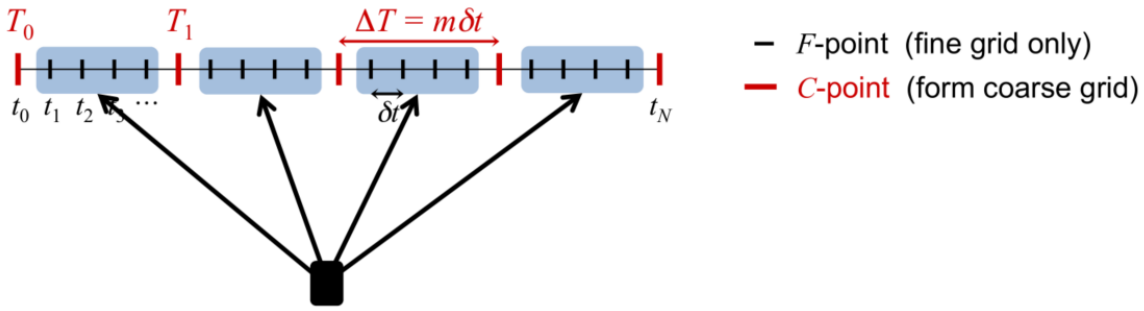


Figure 3 Update all F-point intervals in parallel, using the time propagator Φ.

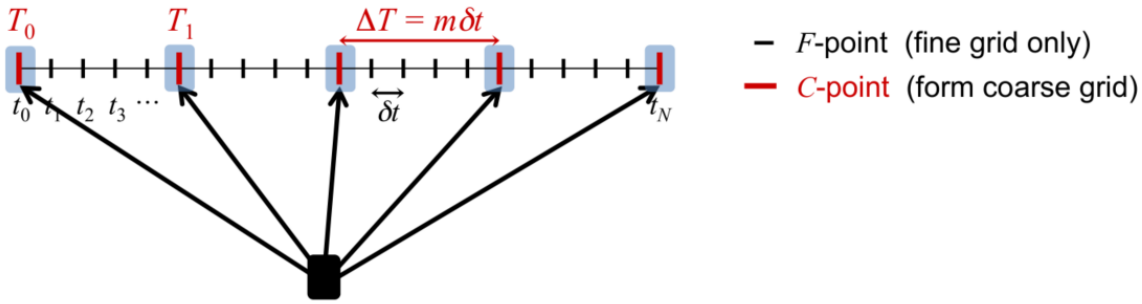


Figure 4 Update all C-points in parallel, using the time propagator Φ.

通常，FCF 和 F 松弛将是指 XBraid 中使用的松弛方法。我们通常认为

- FCF 或者 F 松弛是高度并行
- 但是，存在一个顺序部组，该组点的数量等于两个 C 点之间的 F 点的数量。
- XBraid 使用常规的粗化因子，即 C 点每 m 个点出现一次。

松弛之后，形成粗网格的误差校正。要将数值移动到粗网格，我们

使用限制算子 R，它简单地将 C 点上的值从细网格传递到粗网格。

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

每个 I 之间是 $m-1$ 行。尽管值的传递很简单，但 XBraid 始终在应用 R 之前进行 F 松弛扫描，这等效于使用谐波插值的转置进行限制。另一种解释是，F 松弛将残差压缩到 C 点，即 F 松弛后所有 F 点的残差为 0。因此，传递的限制是有意义的。

为了定义粗网格方程，我们应用 Full Approximation Scheme（FAS）方法，这是多网格的非线性版本。这是为了适应 f 是非线性函数的一般情况。在 FAS 中，猜想解和误差（即 u，$g-Au$）都受到限制。这与线性多重网格相反，线性多重网格通常仅将误差方程限制到粗网格。这种算法上的变化可以解决一般的非线性问题。

应用 FAS 的中心问题是如何形成粗网格矩阵 A_Δ，反过来这又就是如何定义粗网格时间算子 Φ_Δ。最简单的方法(也是最常用的)之一

就是直接令Φ_{Δ}为Φ，但是带有粗网格时间步长$\Delta T=\mathrm{m}\delta t$。例如，如果$\Phi=(\mathrm{I}-\delta\mathrm{tA})^{-1}$对于向后 Euler 法，那么$\Phi_{\Delta}=(\mathrm{I}-\mathrm{m}\delta\mathrm{tA})^{-1}$是一种选择。

有了这个Φ_{Δ}，并且令u_{Δ}是限制在细网格上的解，r_{Δ}是限制在细网格上的误差，粗网格方程

$$A_{\Delta}(v_{\Delta})=A_{\Delta}(u_{\Delta})+r_{\Delta}$$

就被解得。最后，FAS 定义一个粗网格误差估计$\mathrm{e}_{\Delta}=v_{\Delta}-u_{\Delta}$，用$P_{\Phi}$插值回细网格并被添加到修正猜想解。插值等效于将粗网格迭代到细网格上的 C 点,然后进行 F 松弛扫描(即,等效于谐波插值，如上所述)。其中

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

m 是粗化因子。有关 MGRIT 的 FAS 算法的简要说明，请参见双网格算法。

2.3.1　双网格算法

该算法捕获了两层网格的 FAS 过程。使用递归粗网格求解使过程成为多级(即第 3 步成为递归调用)。基于允许误差停止。如果算子是线性的，则此 FAS 循环等效于标准线性多重网格。注意下面我们将 A 表示为函数，而对于线性情况，简化了上述表示法。

1. 使用 FCF 法松弛方程 $A(u)=g$

理解：给时间网格的初始解，分块（一个 C 点之间的间隔）迭代更新 F 点，再更新下一个 C 点，再更新间隔内的 F 点

2. 限制细网格近似解及其误差：

$$\mathrm{u}_{\Delta} \leftarrow R\mathrm{u}，\ \mathrm{r}_{\Delta} \leftarrow R(g-A(u))$$

等价于根据下式更新每个独立时间步

$$u_{\Delta,i} \leftarrow u_{mi}，\ r_{\Delta,i} \leftarrow g_{mi}-A(u)_{mi}, i=0,\ldots,N_{\Delta}$$

3. 求解 $A_{\Delta}(v_{\Delta})=A_{\Delta}(u_{\Delta})+r_{\Delta}$

4. 计算粗网格误差估计：$\mathrm{e}_{\Delta}=v_{\Delta}-u_{\Delta}$

5. 修正：$\mathrm{u} \leftarrow u+P\mathrm{e}_{\Delta}$

这等价于通过把误差加入到 u 在 C 点的值来更新每一个独立的时间节点：

$$u_{mi} = u_{mi} + e_{\Delta,i}$$

然后对 u 进行 F 松弛扫描。

2.3.2 总结

• XBraid 是全局时空问题的迭代求解器

• 用户定义时间迭代算子 Φ，并可以包装现有代码来完成此任务。

• XBraid 的收敛将在很大程度上取决于 Φ_Δ 近似 Φ^m 的效果，这就是 $m\delta t = \Delta T$ 的时间步长大小对于 δt 的时间步长近似于同一时间积分器的 m 个应用的程度。这是研究的主题，但是这种近似不需要捕获细尺度的行为，而是通过在细网格上的松弛来捕获。

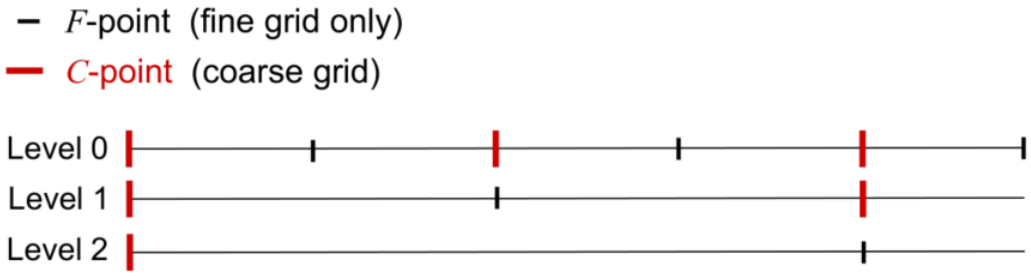
• 粗网格被顺序地精确求解，这可能是 Parareal 这种两级方法的一个瓶颈，但对于类似 XBraid 的多级算法却是没有的，因为最粗糙的网格的规模很小。

• 通过将粗网格形成为具有与细网格相同的稀疏结构和时间步长，该算法可以轻松、高效地重现。

• 插值是理想的还是精确的，因为插值的应用在所有 F 点上都

留下零残差。

• 递归应用该过程，直到达到平凡的时间网格为止，例如，2 或者 3 个时间点。因此粗化因子 m 决定在所有阶层有多少个等级。例如如下图，是 3 等级阶层。因为有 6 个点且 $m=2, m^2<6\le m^3$ 所以选择 3 等级。如果粗化因子 $m=4$，那么只要 2 等级。



默认情况下，XBraid 会将时域 细分为大小均匀的时间步长。XBraid 的结构可以处理可变的时间步长和自适应的时间步长。

2.4 XBraid 代码概述

XBraid 旨在与可以按我们的界面包装的现有应用程序代码一起运行。该应用程序代码将实现一些时间方程行进模拟，例如流体流动。本质上，用户必须获取其应用程序代码并提取独立的时间步长函数 Φ，这个函数可以将解从一个时间值演变为另一个时间值，而与时间步长无关。在这个完成之后，XBraid 只关

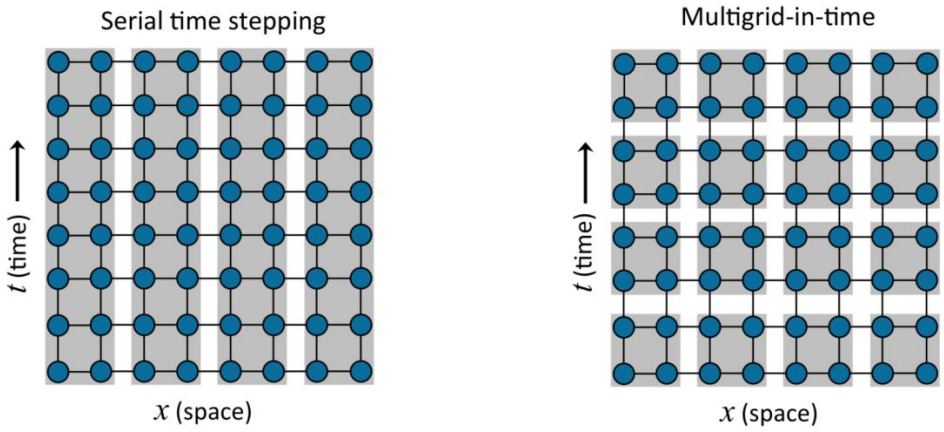
心在时间域上的并行。

XBraid

- C 语言编写，并且带有 Fortran 和 C++接口
- 对于并行使用 MPI
- 通过源代码的中的注释和.md 文件自行编写文档
- 函数和结构体以 braid 为前缀
  - -用户路径以 braid_为前缀
  - -开发者路径以_braid_为前缀

2.4.1 并行分解和存储

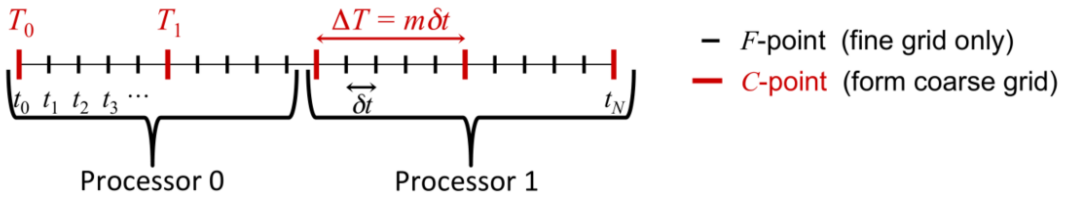
XBraid 并行分解问题如下图。



很显然看出，传统时间迭代同一时间点内只存储一个时间步，但只有空间数据分解和空间并行性。另一方面，XBraid 同时存

储多级时间步，每个处理器都有一个时空块，反映了时空并行性。

• XBraid 仅处理时间并行性，而与空间分解无关。每个处理器拥有一定数量的 CF 点间隔。在下图中，处理器 1 和处理器 2 每一个都拥有 2 个 CF 间隔。XBraid 在最细的网格上平均分配间隔。



• XBraid 显着提高了并行性，但是现在需要存储几个时间步骤，这需要更多的内存。XBraid 采用两种策略来解决增加的内存成本。

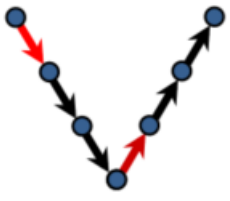
1. 不必立即解决整个问题。存储一个时空块即可。也就是说，用所有可用内存解决时间步长（例如 k 个时间步长）。然后继续进行下 k 个时间步。
2. XBraid 提供仅存储 C 点的支持。每当需要 F 点时，都会通过 F 松弛生成它。更准确地说，仅存储上图中的

红色 C 点时间值。当 $m=8,16,32,\cdots,$ 粗化通常比较激进，因此与存储所有时间值相比，XBraid 的存储要求会大大降低。

总体而言，如果使用时空粗化（请参阅最简单的示例），则使用 XBraid 时每个处理器的内存乘数为 $O(1)$，而对于仅进行时间粗化则为 $O(log_m N)$。“仅时间粗化”选项是默认选项，不需要用户编写的空间插值/限制代码（对于空时协同工作就是这种情况）。我们注意到对数的底是 m，它可能很大。

2.4.2 循环和松弛策略

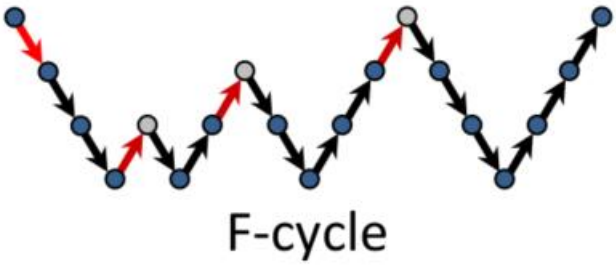
XBraid 中有两种主要的循环策略：F 循环和 V 循环。这两个循环的不同之处在于访问粗化级别的频率和顺序不同。接下来描述一个 V 循环，它是“双网格”算法的简单递归应用。



V-cycle

F 循环以不同的顺序更频繁地访问粗网格。本质上，F 循环将

V 循环用作后平滑器，这是松弛的昂贵选择。但是，这项额外的工作使您更接近于双网格循环，并且以更多的工作为代价，加快了收敛速度。在 figure 2 中可以看到一个 V 循环作为松弛方案的有效性，其中一个 V 循环全局传播并消除了误差。接下来描述 F 循环的循环策略。



接下来，我们对 F 循环与 V 循环进行几点讨论。

• 一个 V 循环迭代比一个 F 循环迭代简单。

• 但是，F 周期通常收敛得更快。对于某些测试用例，这种差异可能会很大。最佳解决时间的循环选择取决于问题。有关循环策略的案例研究，请参见本示例的比例研究。

• 对于异常强大的 F-循环，可以将选项 braid_SetNFMGVcyc 设置为使用多个 V 循环作为松弛。事实证明，这对于强对流性质的某些问题很有用。

FC 松弛扫描的数量是另一个重要的算法设置。注意到至少一个 F 松弛扫描始终在一个级别上进行。有关松弛的一些总结点如下。

• 使用 FCF，FCFCF 或 FCFCFCF 松弛对应于分别向 braid_SetNRelax 传递 1、2 或 3 的值，并且将导致 XBraid 循环随着松弛数量的增加而更快地收敛。

• 但是，随着松弛次数的增加，每个 XBraid 周期变得更加昂贵。最佳求解时间的最佳松弛策略将取决于问题。

• 但是，一个好的第一步是在所有级别上尝试 FCF（即 braid_SetNRelax（core，-1，1））。

3. 常见的最优方法是先在所有级别上设置 FCF（即 braid_setnrelax（core，-1，1）），然后在级别 0 上覆盖 FCF 选项，以便仅在级别 0 上完成 F 松弛（即 braid_setnrelax（core，0、1））。另一种策略是在所有级别上与 F 循环一起使用 F 松弛。

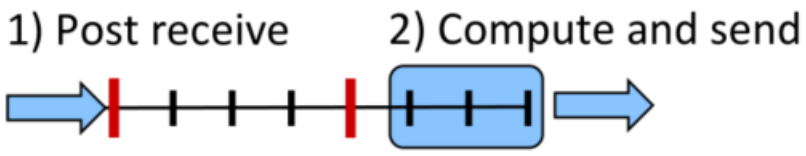
4. 看 Scaling Study with this Example 参见本示例的比例

研究。

最后，Parallel Time Integration with Multigrid 对于循环和松弛策略有一个更深的研究。

2.4.3 重叠通信和计算

XBraid 有效地重叠了通信和计算。XBraid 的主要计算核心是一次松弛扫描，涉及所有 CF 间隔。在松弛扫描开始时，每个进程首先在其最左边的位置非阻塞接收。然后，它从最右边的间隔开始在每个间隔中执行 F 松弛，以尽快将数据发送到相邻进程。如果每个进程在此 XBraid 级别具有多个 CF 间隔，则该策略允许完全重叠。



2.4.4 配置 XBraid 层次结构

一些更基本您可以控制的 XBraid 函数调用在此处讨论

braid_SetFMG: switches between using F- and V-cycles.

braid_SetMaxIter: sets the maximum number of XBraid iterations

braid_SetCFactor: sets the coarsening factor for any (or all levels)

braid_SetNRelax: sets the number of CF-relaxation sweeps for any (or all levels)

braid_SetRelTol, braid_SetAbsTol: sets the stopping tolerance

braid_SetMinCoarse: sets the minimum possible coarse grid size

braid_SetMaxLevels: sets the maximum number of levels in the XBraid hierarchy

2.4.5 停止误差

另一个重要的配置方面是关于设置剩余的停止误差。设置误差

涉及以下三个 XBraid 选项：

1. braid_PtFcnSpatialNorm

用户定义的函数通过采用 braid_Vector 的范数来执行空间范数。常见的选择是标准的欧几里得范数（2-范数），但许多其他选择也是可能的，例如基于有限元素空间的 L2-范数。

2. braid_SetTemporalNorm

此选项确定如何获取全局时空误差范数。也就是说，这决定了如何组合每个时间步长的 braid_PtFcnSpatialNorm 返回的空间范数，以获得关于空间和时间的全局范数。然后，这是控制停顿的全局范数。braid_SetTemporalNorm 支持三个 *tnorm* 选项。我们将总索引 i 定义为精细时间网格上的所有 C 点值，k 表示当前 XBraid 迭代，r 为残差值，space_time 范数为整个时空域的范数，而 space_norm 为用户-从 braid_Pt FcnSpatialNorm 定义空间范数。因此，r_i 是在第 i 个 C 点的残差，而 $r^{(k)}$ 是在第 k 次 XBraid 迭代时的残差。然后将这三个选项定义为：

tnorm=1：空间范数的一范数求和

$$\|\mathrm{r}^{(k)}\|_{space\_time}=\sum_i\|r_i^{(k)}\|_{spatial\_rom}$$

如果 braid_PtFcnSpatialNorm 是空间的一范数，则这等效于全局时空残差矢量的一范数。

$tnorm=2$：空间范数的二范数求和

$$\|\mathrm{r}^{(k)}\|_{space\_time}=(\sum_i\|r_i^{(k)}\|_{spatial\_rom}^2)^{1/2}$$

如果 braid_PtFcnSpatialNorm 是空间上的欧几里得范数（两个范数），则这等效于全局时空残差矢量的欧几里得范数。

$tnorm=3$：空间范数的无穷范数组合

$$\|\mathrm{r}^{(k)}\|_{space\_time}=\max_{\mathrm{i}}\|\mathrm{r}_{\mathrm{i}}^{(k)}\|_{spatial\_norm}$$

如果 braid_PtFcnSpatialNorm 是空间上的无穷范，则这等效于全局时空残差矢量的无穷范。

默认的选择$tnorm=2$

3. braid_SetAbsTol, braid_SetRelTol

如果使用绝对残差

$$\|\mathrm{r}^{(k)}\|_{space\_time}<tol$$

定义何时停止

如果使用相对残差

$$\frac{||\mathrm{r}^{(k)}||_{space\_time}}{||\mathrm{r}^{(0)}||_{space\_time}} < tol$$

定义何时停止。即在与停止残差比较之前，通过初始残差缩放当前的第 k 个残差。这类似于在空间多重网格中使用的典型相对剩余停止残差，但在这种情况下可能是危险的选择。

选择停止残差时应格外小心。例如，如果使用相对残差，则对于大量时间步长，初始猜想解为零时，可能会出现问题。以所有时间值 t> 0 的初始猜想解（由 braid_PtFcnInit 定义）为 0 的情况为例，初始残差范数实际上只会在第一个时间值处为非零，

$$||\, r^{(0)}\,||_{space\_time} \approx ||\, r_1^{(k)}\,||_{spatial\_norm}$$

这将使相对的停止残差产生偏差，尤其是在时间步长增加的情况下，而初始剩余范数则不会。

更好的策略是选择一个考虑到您的时空域大小的绝对容差，如

本示例的 Scaling Study with this Example 中所述，或使用无穷范数时间范数选项。

2.4.6 Debugging XBraid

使用 XBraid 包装和调试代码通常需要遵循几个步骤。

使用 XBraid 测试功能测试包装的功能，例如 braid_TestClone 或 braid_TestSum。

将最大级别设置为 1（braid_SetMaxLevels）并运行 XBraid 模拟。您应该获得与顺序时间迭代所获得的完全相同的答案。

如果您确保 XBraid 和顺序时间迭代所使用的时间网格是逐位相同的（通过使用用户定义的时间网格选项 braid_SetTimeGrid），那么它们的解决方案的协议应该逐位相同。

继续使用等于 1 的最大级别，但要及时切换到两个处理器。再次检查答案是否与连续时间迭代完全匹配。该测试检查 braid_Vector 中的信息是否足以及时在第二个处理器上正确启动仿真。

将最大级别设置为 2，将终止公差设置为 0.0（braid_SetAbsTol），将最大迭代次数设置为 3（braid_SetMaxIter），然后打开选项 braid_SetSeqSoln。这将使用顺序时间迭代的解决方案作为 XBraid 的初始猜测，然后运行 3 次迭代。每次迭代的残差都应该正好为 0，从而验证 XBraid 的定点性质和正确的实现（希望如此）。残差可以约为机器 ε （或更小）。及时对多个处理器重复此测试（如果可能的话，还需要空间）。

类似的测试通过将 3 的打印级别传递给 braid_SetPrintLevel 来打开调试级别的打印。这将打印出每个 C 点的剩余范数。具有 FCF 松弛的 XBraid 具有以下特性：每次迭代将精确的解决方案传播到两个 C 点。因此，这应该通过头这么多时间点的数字零残差值来反映。及时对多个处理器重复此测试（如果可能的话，还需要空间）。

最后，运行一些多级测试，确保 XBraid 结果在按顺序的时间步长生成的解决方案的极限公差之内。及时对多个处理器重复此测试（如果可能的话，还需要空间）。

恭喜！您的代码现已验证。

2.5 使用 XBraid_Adjoint 计算导数

XBraid_Adjoint 与德国 TU Kaiserslautern 的科学计算小组合作开发，特别是与 Stefanie Guenther 博士和 Nicolas Gauger 教授合作开发。

在许多应用场景中，ODE 系统由一些独立的设计参数 ρ 驱动。这些可以是唯一确定 ODE 解的任何时间相关或时间无关参数（例如边界条件，材料系数等）。

在离散的 ODE 设置中，可以编写用户的时间步长为

$$u_i = \Phi_i(u_{i-1}, \rho), \forall i = 1, \ldots, N,$$

Φ_i 传递 u_{i-1} 在时间 t_{i-1} 到下一个时间步 t_i，现在也依赖于独立的设计参数 ρ。为了量化给定设计的仿真输出，可以建立一个实值目标函数，以测量 ODE 解决方案的质量：

$$J(u, \rho) \in R$$

这里，$u = (u_0, \cdots, u_N)$ 表示给定设计的时空状态解决方案。

XBraid_Adjoint 是 XBraid 的一致离散时间并行伴随求解器，它

提供了输出量 J 的用户定义的设计参数 ρ 的灵敏度信息。计算灵敏度的能力可以极大地改善和增强仿真工具，例如用于求解

- 设计优化问题，
- 最佳控制问题
- 用于验证和验证目的的参数估计
- 误差估计
- 不确定度量化技术

XBraid_Adjoint 相对于伴随时间步长方案是非侵入性的，因此可以通过扩展的用户界面轻松集成现有的时间序列伴随代码。

2.5.1 基于伴随的灵敏度计算简介

通过求解其他所谓的伴随方程，基于伴随的灵敏度可计算 J 相对于设计参数 ρ 的变化的总导数。接下来，我们将简要介绍该想法。如果您通常熟悉伴随灵敏度计算，则可以跳过本部分，并立即转到 XBraid_Adjoint 算法概述。关于伴随方法的信息可以在 [Giles，Pierce，2000][3] 中找到。

考虑一个增强的（所谓的拉格朗日）函数

$$L(u,\rho)=J(u,\rho)+\bar{u}^{T}A(u,\rho)$$

离散时间步长 ODE 方程在

$$A(u,\rho)=\begin{pmatrix}\Phi_1(u_0,\rho)-u_1\\ \vdots\\ \Phi_N(u_{N-1},\rho)-u_N\end{pmatrix}$$

已添加到目标函数，并与所谓的伴随变量$\bar{u}=(\bar{u}_1,\ldots,\bar{u}_N)$相乘。由于对于满足离散 ODE 方程的所有设计和状态变量，相加项均为零，因此 J 和 L 相对于设计的总导数匹配。使用微分链规则，该导数可以表示为

$$\frac{dJ}{d\rho}=\frac{dL}{d\rho}=\frac{\partial J}{\partial\rho}\frac{du}{d\rho}+\bar{u}^{T}\left(\frac{\partial A}{\partial u}\frac{du}{d\rho}+\frac{\partial A}{\partial\rho}\right)$$

其中∂表示偏导数 - 与 d 表示的总导数（即灵敏度）相反

在计算此导数时，红色项是计算上最昂贵的项。实际上，计算这些灵敏度的成本与设计参数的数量（即 ρ 的维数）成线性比例。这些成本会快速增长。例如，考虑一个有限差分设置，其中每个设计变量都需要重新计算整个时空状态，因为必须在设计空间的所有单位方向上计算设计的扰动。为了避免这些费用，伴随方法旨在设定伴随变量$\bar{u}$，使得这些红色项在上述表达式中加起来为

零。因此，如果我们首先解决

$$\left(\frac{\partial J}{\partial \mathrm{u}}\right)^{T}+\left(\frac{\partial A}{\partial u}\right)^{T}\bar{u}=0$$

对于伴随变量 $\bar{u}$ ，那么 J 的所谓减小的梯度，即 J 的总导数相对于设计的转置，由下式给出

$$\left(\frac{dJ}{d\rho}\right)^{T}=\left(\frac{\partial J}{\partial \rho}\right)^{T}+\left(\frac{\partial A}{\partial u}\right)^{T}\bar{u}$$

该策略的优势在于，为了计算 J 对 ρ 的灵敏度，除了评估偏导数外，还需要求解 $\bar{\mathrm{u}}$ 的一个附加时空方程（伴随）。因此，在这种设置下，计算 $\mathrm{dJ/d}\rho$ 的计算成本不会随着设计参数的数量而增加。

对于时间相关的离散 ODE 问题，上面的伴随方程式如下：

不稳定伴随： $\bar{u}_i=\partial_{ui}J(u,\rho)^{T}+(\partial_{ui}\Phi_{i+1}(u_i,\rho))^{T}\bar{u}_{i+1},\forall i=1,\ldots,N$

使用终止条件 $u_{N+1}:=0$，降低的梯度由下式给出

下降梯度： $\left(\frac{\partial J}{\partial \rho}\right)^{T}=\partial_{\rho}J(u,\rho)^{T}+\sum_{i=1}^{N}(\partial_{\rho}\Phi_i(u_{i-1},\rho))^{T}\bar{u}_i$

2.5.2 XBraid_Adjoint 算法概述

从最终条件 $\bar{u}_{N+1}=0$ 开始，原则上可以以时间序列的方式“按时间

倒退”求解非稳态伴随方程。但是，时间并行 XBraid_Adjoint 求解器通过将时间倒退阶段沿时域分布到多个处理器上来提供加速。它的实现基于应用于一个原始 XBraid 迭代的自动求逆的反向模式技术。为此，通过目标函数评估来扩展每个原始迭代，然后更新时空伴随变量 $\bar{u}$，并评估由 ρ 表示的缩减梯度。特别是，执行以下所谓的 piggy-back 迭代：

1. XBraid：更新状态并评估目标函数

$$u^{(k+1)} \leftarrow XBraid(u^{(k)}, \rho), J \leftarrow J(u^{(k)}, \rho)$$

2. XBraid_Adjoint：更新伴随并评估减小的梯度

$$\bar{u}^{(k+1)} \leftarrow XBraid\_Adjoint(u^{(k)}, \bar{u}^{(k)}, \rho), \bar{\rho} \leftarrow \left(\frac{dJ(u^{(k)}, \rho)}{dp} \right)^T$$

每个 XBraid_Adjoint 迭代都会通过原始 XBraid 多重网格循环向后移动。它以相反顺序收集基本 XBraid 运算的局部偏导数，并使用微分链规则将它们连接起来。这是自动微分（AD）反向模式的基本思想。这将产生一个一致的离散时间并行伴随求解器，该求解器继承了原始 XBraid 求解器的并行缩放属性。

此外，XBraid_Adjoint 对基于顺序时间行进方案的现有伴随方法

不具有干扰性。它将附加的用户定义的例程添加到原始 XBraid 接口，以定义正向步进器向后传播的敏感度的传播，以及在每个时间步长评估局部目标函数的偏导数。在时间序列不稳定的伴随求解器已经可用的情况下，可以根据伴随用户界面轻松包装这种向后的时间步进功能，而无需额外的编码。

上述 piggy-back 迭代中的伴随解以与原始状态变量相同的收敛速度收敛。但是，由于伴随方程式取决于状态解，因此伴随收敛将稍微滞后于状态收敛。有关 XBraid_Adjoint 收敛结果和实现细节的更多信息，请参见[Gunther，Gauger，Schroder 2017]。

2.5.3 XBraid_Adjoint 代码概述

XBraid_Adjoint 提供了一种非侵入性方法来对现有时间序列伴随代码进行时间并行化。为此，扩展的用户界面允许用户将其现有代码包装起来，以评估目标功能并根据 XBraid_Adjoint 接口将时间倒退的伴随步骤执行到例程中。

2.5.3.1 目标函数评估

XBraid_Adjoint 的用户界面允许使用以下类型的目标函数：

$$J=\mathrm{F}\left(\int_{t_0}^{t_1} f(u(t),\rho)dt\right).$$

这涉及一些与时间有关的感兴趣量 f 的时间积分部分以及后处理函数 F 。可以使用选项 braid_SetTStartObjective 和 braid_SetTStopObjective 设置时间间隔边界 t_0,t_1 ，否则将考虑整个时域。请注意，这些选项可用于仅通过设置 $t_0=t_1$ 在一个特定时间实例进行评估的目标函数（例如，在仅关注最后一个时间步的情况下）。后处理功能 F 提供了进一步修改时间积分的可能性，例如，用于设置跟踪型目标函数（减去目标值和平方），或者添加松弛或惩罚条件。虽然对于 XBraid_Adjoint 必须定义 f ，但后处理例程 F 是可选的，并通过可选的 braid_SetPostprocessObjective 和 braid_SetPostprocessObjective_diff 例程传递给 XBraid_Adjoint。XBraid_Adjoint 将通过在求和给定时域中的 f 评估来执行时间积分

$$I\leftarrow\sum_{\mathrm{i}=i_0}^{i_1} f(u_i,\rho)$$

然后调用后处理函数F，如果设置：

$$J \leftarrow F(I,\rho)$$

请注意，任何用于计算 I 的集成规则，例如用于缩放$f(\ )$的贡献，必须由用户完成。

用户例程的偏导数

用户需要提供 X←Braid_Adjoint 的时间步长Φ的导数和函数评估f（可能还有F）。通过转置矩阵向量乘积以下列方式提供这些：

1. 目标函数J的导数：

• 与时间有关的部分f：用户提供一个例程，该例程评估f的以下转置偏导数乘以标量输入F：

$$\bar{\mathrm{u}}_i \leftarrow \left(\frac{\partial f(u_i,\rho)}{\partial u_i}\right)^T \bar{F}$$

$$\bar{\rho} \leftarrow \bar{\rho} + \left(\frac{\partial f(u_i,\rho)}{\partial \rho}\right)^T \bar{F}$$

如果未设置后置处理函数F，则标量输入$\bar{F}$等于 1.0。

• 后处理 F：如果设置了后处理例程，则用户需要通过以下方式提供它的转置偏导数：

$$\bar{F} \leftarrow \frac{\partial F(I,\rho)}{\partial I}$$

$$\bar{\rho} \leftarrow \rho + \frac{\partial F(I,\rho)}{\partial \rho}$$

2. 时间步长Φ_i的导数：用户提供了一个例程，用于计算Φ_i的以下转置偏导数乘以伴随的输入向量$\bar{u}_i$：

$$\bar{u}_i \leftarrow \left(\frac{\partial \Phi(u_i,\rho)}{\partial \rho}\right)^T \bar{u}_i$$

$$\bar{\rho} \leftarrow \bar{\rho} + \left(\frac{\partial \Phi(u_i,\rho)}{\partial \rho}\right)^T \bar{u}_i$$

注意，关于ρ的偏导数总是更新减小的梯度ρ而不是覆盖它（即，它们是一个+等于运算，+ =）。因此，需要在每次 XBraid_←Adjoint 迭代之前将梯度重置为零，这由 XBraid_Adjoint 调用附加的用户定义例程 braid_PtFcnResetGradient 来解决。

根据设计变量的性质，有必要在 XBraid_Adjoint 完成后从所有时间处理器中收集ρ中的梯度信息。如果需要，用户有责任这样做，例如通过调用 MPI_Allreduce。

停止公差

类似于原始 XBraid 算法，用户可以基于伴随残差范数为

XBraid_Adjoint 选择暂停公差。一个绝对公差

（braid_SetAbsTolAdjoint）

$$\| \bar{u}^{(k)} - \bar{u}^{(k-1)} \|_{space\_time} < tol\_adjoint$$

或相对公差（braid_SetRelTolAdjoint）

$$\frac{\| \bar{u}^{(k)} - \bar{u}^{(k-1)} \|_{space\_time}}{| \bar{u}^{(1)} - \bar{u}^{(0)} \|_{space\_time}} < tol\_adjoint$$

能被选择。

有限差分测试

您可以使用有限差分来验证从 XBraid_Adjoint 计算出的梯度。令 e_i 表示设计空间中的第 i 个单位向量，则梯度的第 i 个条目应与

第 i 个有限差分：$\frac{J(u_{\rho+he_i}, \rho+he_i) - J(u,\rho)}{h}$

小扰动 $h>0$。在这里，$u_{\rho+he_i}$ 表示扰动的设计变量的新状态解。请记住，对于很小的扰动 $h \to 0$，计算有限差分时必须考虑舍入误差。因此，您应该更改参数以找到最合适的参数。

为了在计算扰动的目标函数值时节省一些计算工作，XBraid_Adjoint 可以在 ObjectiveOnly 模式下运行，请参见 braid_SetObjectiveOnly。在这种模式下，XBraid_Adjoint 将仅求解 ODE 系统并评估目标函数，而无需实际计算其导数。此选项在优化框架内可能也很有用，例如用于实施寻线程序。

入门

• 查看简单示例 Simplest XBraid_Adjoint 示例，以开始使用。此示例在 examples / ex-01-adjoint.c 中，该示例实现了标量 ODE 的 XBraid_Adjoint 灵敏度计算。