Revisionist integral deferred correction methods are a family of parallel-in-time methods to solve systems of initial values problems. The approach is able to bootstrap lower-order time integrators to provide high-order approximations in approximately the same wall-clock time, hence providing a multiplicative increase in the number of compute cores utilized. Here we provide a library that automatically produces a parallel-in-time solution of a system of initial value problems given user-supplied code for the right-hand side of the system and a sequential code for a first-order timestep. The user-supplied timestep routine may be explicit or implicit and may make use of any auxiliary libraries that take care of the solution of any nonlinear algebraic systems that may arise or the numerical linear algebra required.

Categories and Subject Descriptors: G.1.0 [Numerical Analysis]: Parallel Algorithms; G.1.7 [Numerical Analysis]: Ordinary Differential Equations—Initial-value problems

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Parallel in time, deferred correction

ACM Reference Format:
Benjamin W. Ong, Ronald D. Haynes, and Kyle Ladd. 2016. Algorithm 965: RIDC methods: A family of parallel time integrators. ACM Trans. Math. Softw. 43, 1, Article 8 (August 2016), 13 pages. DOI: http://dx.doi.org/10.1145/2964377

1. INTRODUCTION

The fast, accurate solution of an initial-value problem (IVP) of the form

\[ y'(t) = f(t, y), \quad y(0) = y_0, \quad t \in [0, T]. \]

(1)

where \( y(t) \in \mathbb{C}^N \), \( f : [\mathbb{R} \times \mathbb{C}^N] \rightarrow \mathbb{C}^N \) is of practical interest in scientific computing.

IVP (1) often arises from the spatial discretization of partial differential equations and may require either an explicit or implicit time integrator. The purpose of this software is to "wrap" a user-implemented first-order explicit or implicit solver for IVP (1) into a high-order parallel solver—that is, given \( (t_n, y_n, f_n) \), a user specifies a function that returns \( (t_{n+1}, y_{n+1}, f_{n+1}) \) using either a forward Euler or backward Euler integrator. This work differs from existing ODE integration software or libraries in which a user typically only needs to specify the system of ODEs and relevant problem parameters.

This work was supported by AFOSR grant FA9550-12-1-0455, an NSERC Discovery grant (Canada), and the Institute for Cyber-Enabled Research at Michigan State University. Superior, a high-performance computing cluster at Michigan Technological University, was used in part to obtain results presented in this publication.

Authors' addresses: B. W. Ong, Michigan Technological University, Department of Mathematical Sciences, Houghton, MI, 49931, 1-906-487-3367; email: ongbw@mtu.edu; R. D. Haynes, Memorial University, Department of Mathematics and Statistics, St. John's, NL, Canada A1C 5S7, 709-864-8825; email: rhaynes@mun.ca; K. Ladd, Amazon.com, 1918 8th Ave, Seattle WA 98108-1226, 734-560-0703; email: laddkyle@amazon.com. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 0098-3500/2016/08-ART8 $15.00
DOI: http://dx.doi.org/10.1145/2964377
The upside is that our software provides a parallel-in-time solution while giving the user complete control of the first-order timestep routine. For example, users may choose their own quality libraries for the solution of systems of nonlinear algebraic equations or efficient linear system solvers particularly tuned to the structure of their problems.

There are three general approaches for a time-parallel solution of IVPs [Burrage 1993]. One approach is *parallelism-across-the-problem*, where a problem is decomposed into subproblems that can be computed in parallel and an iterative procedure is used to couple the subproblems. Examples of this class of methods include parallel waveform relaxation methods [Vandewalle and Roose 1989]. The second approach is *parallel-across-the-step* methods, in which the time domain is partitioned into smaller temporal subdomains that are solved simultaneously. Examples of this class of methods include parareal methods [Maday and Turinici 2002; Gander and Vandewalle 2007], in which the method alternates between applying a coarse sequential solver and a fine parallel solver. The third approach is *parallelism-across-the-method*, in which one exploits concurrent function evaluations within a step to generate a parallel time integrator. This approach typically allows for small-scale parallelism, constrained by the number of function evaluations that can be evaluated in parallel. This is often related to the order of the approximation. Examples of Runge-Kutta methods where stages can be evaluated in parallel include Miranker and Liniger [2967], Enenkel and Jackson [1997], and Ketcheson and Bin Waheed [2014]. Alternatively, one can use a predictor-corrector framework to generate parallel-across-the-method time integrators. This includes parallel extrapolation methods [Kappeller et al. 1996], as well as revisionist integral deferred correction (RIDC) integrators [Christlieb et al. 2010b; Christlieb and Ong 2011], which are the focus of this article. A survey of parallel time integration methods has recently appeared [Gander 2015].

1.1. Related Software

There are several well-established software packages for solving differential algebraic equations; however, not many of them are able to solve IVPs (1) in parallel. For sequential integrators, probably the most well known are MATLAB routines *ode45*, *ode23*, *ode15s* [Shampine et al. 1999] to solve their systems of differential equations. These schemes use embedded RK pairs or numerical differentiation formulas (of the specified order) to approximate solutions to the differential equations using adaptive timesteping. Readers might also be familiar with DASSL [Petzold 1983], which implements backward differentiation formulas of orders one through five. The nonlinear system at each timestep is solved by Newton’s method, and the resulting linear systems are solved using routines from LINPACK. DASSL leverages the SLATEC Common Mathematical Library [Vandevender and Haskell 1982] for step-size adaptivity. Also popular are ODEPACK [Hindmarsh 1983] and VODE [Brown et al. 1989], a collection of Fortran solvers for IVPs, and SUNDIALS, a suite of robust time integrators and nonlinear solvers [Hindmarsh et al. 2005], and there are a variety of ODE and DAE timesteppers implemented in PETSc [Balay et al. 2014] and GSL [Gough 2009].

The selection of parallel solvers for IVPs is fairly sparse. EPPEER [Schmitt 2013] is a Fortran95/OpenMP implementation of explicit parallel two-step peer methods [Weiner et al. 2008] for the solution of ODEs on multicore architectures. PyPFASST [Emmett 2013] is a Python implementation of a modified parareal solver for ODEs and PDEs [Emmett and Minion 2012]. XBRAID [Schroder et al. 2015] is a C library that implements a multigrid-reduction-in-time algorithm [Falgout et al. 2014], in which multiple time grids of different granularity are distributed across processors using MPI. PFASST++ [Emmett et al. 2015] is a C++ implementation of the parallel full approximation scheme in space and time (PFASST) algorithm [Emmett and Minion 2014]. There are other implementations (e.g., the dependency-driven parareal framework
developed at Oakridge National Laboratory [Elwasif et al. 2011]) that do not appear to be available for download at present.

2. REVIEW OF RIDC METHODS

Spectral deferred correction (SDC) [Dutt et al. 2000] provides an iterative correction of an approximate solution by solving an integral formulation of an error equation. This integral form stabilizes the classical differential deferred correction approach. RIDC is a reformulation of SDC, pipelining successive calculations so that corrections can be obtained in parallel with an appropriate time lag. SDC, in contrast, is a sequential algorithm. Unlike SDC, which uses Gauss-Lobatto nodes, RIDC uses uniformly spaced nodes to minimize the memory footprint and to allow one to embed high-order integrators [Christlieb et al. 2009, 2010a].

The basic idea of the IDC and RIDC approaches is to formulate associated error IVPs that correct numerical errors from the solutions to IVP (1); the parallelism arises from the ability to simultaneously compute solutions to both IVP (1) and solutions to the associated error IVPs. In this section, we review the formulation of the error equations, discretizations, and parallel properties of the RIDC algorithm. Please refer to Christlieb et al. [2010b] and Christlieb and Ong [2011] for accuracy and stability properties of the RIDC approach.

2.1. Error IVPs

Denote the (unknown) exact solution of IVP (1) as \( y(t) \), and the approximate solution as \( u(t) \), with \( u(0) = y(0) \). The error in the approximate solution is \( e(t) = y(t) - u(t) \). Define the residual (sometimes known as the defect) as \( r(t) = u'(t) - f(t, u) \). Then, the time derivative of the error satisfies

\[
e'(t) = y'(t) - u'(t) = f(t, u + e) - f(t, u) - r(t).
\]

Since \( e(0) = u(0) - y(0) = 0 \), we have just derived the associated error IVP. For stability, the integral form of the error IVP is preferred [Dutt et al. 2000]:

\[
\left( e + \int_0^t r(\tau) d\tau \right)' = f(t, u + e) - f(t, u).
\] (2)

Observing that the corrected approximation \( u + e \) is still an approximation if the error equation (2) is solved numerically, we adopt a more general notation that will allow us to iteratively correct the solution until a desired accuracy is reached. Denote the initial approximation as \( u^{[0]} \), the \( p \)th approximation as \( u^{[p]} \), and the error to \( u^{[p]} \) as \( e^{[p]} \). Then, the error equation can be rewritten as

\[
\left( e^{[p]} + \int_0^t r^{[p]}(\tau) d\tau \right)' = f(t, u^{[p]} + e^{[p]}) - f(t, u^{[p]}),
\] (3)

where \( r^{[p]} = u^{[p]}(t)' - f(t, u^{[p]}) \).

2.2. Discretization

With some algebra, a first-order explicit discretization of (3), written in terms of the solution, gives

\[
u^{[p+1]}_{n+1} = u^{[p+1]}_n + \Delta t f(t_n, u^{[p+1]}_n) - \Delta t f(t_n, u^{[p]}_n) + \int_{t_n}^{t_{n+1}} f(\tau, u^{[p]}_n) d\tau.
\] (4)
Likewise a first-order implicit discretization of (3) gives
\[ u_{n+1}^{[p+1]} = u_{n+1}^{[p]} + \Delta t f(t_{n+1}, u_{n+1}^{[p+1]}) - \Delta t f(t_{n+1}, u_{n+1}^{[p]}) + \int_{t_n}^{t_{n+1}} f(\tau, u^{[p]}) \, d\tau. \] (5)

In both semidiscretizations (4) and (5), a sufficiently accurate quadrature is needed to approximate the integrals present [Dutt et al. 2000]. If a first-order predictor was applied to obtain an approximate solution to (1), and first-order correctors such as (4) and (5) are used, approximating the quadrature using

\[
\int_{t_n}^{t_{n+1}} f(\tau, u^{[p]}) \, d\tau \approx \begin{cases} 
\frac{1}{\alpha_{p\nu}} \left( t - t_{n+1-i} \right), & \text{if } n \geq p, \\
\frac{1}{\alpha_{p\nu}} \left( t_{\nu} - t_i \right), & \text{if } n < p,
\end{cases}
\]

where \( \alpha_{p\nu} \) are quadrature weights,

\[
\alpha_{p\nu} = \begin{cases} 
\int_{t_n}^{t_{n+1}} \prod_{i=0, i \neq \nu}^{p+1} \frac{1}{\left( t_{n+1-i} - t_{n+1-i} \right)} \, dt, & \text{if } n \geq p, \\
\int_{t_n}^{t_{n+1}} \prod_{i=0, i \neq \nu}^{p+1} \frac{1}{\left( t_{\nu} - t_i \right)} \, dt, & \text{if } n < p,
\end{cases}
\]

results in a \( P \)th order method, if \((P - 1)\) such corrections are applied.

2.3. Stability
A study of the (linear) stability of explicit RIDC methods is provided in Christlieb et al. [2010b] and for implicit RIDC methods in Christlieb and Ong [2011]. The results indicate that the region of absolute stability of RIDC methods approach the region of absolute stability of the underlying predictor as the number of timesteps increases. Moreover, the implicit RIDC4-BE method preserves the A-stability property of backward Euler.

2.4. Parallelization
As mentioned earlier, the parallelism arises from the ability to simultaneously compute solutions to both IVP (1) and solutions to the associated error IVPs (3). This is possible if there is some staggering to decouple solutions of IVP (1) and the error equations. As shown in Figure 1, staggering of one timestep is required to compute solutions in a pipeline-parallel fashion. For example, while the predictor computes a solution at time \( t_{10} \), the first corrector computes the correction at time \( t_9 \), the second corrector the second correction at time \( t_8 \), and so on. We discuss the memory footprint and the start-up routine required by the RIDC method in Section 2.5 before presenting a pseudoalgorithm for the RIDC methods.

2.5. Memory Footprint, Efficiency, Start-Up, and Shutdown
Figure 1 also shows the memory footprint required to execute the RIDC method in a pipeline-parallel fashion. The memory footprint is copies of the solution vector evaluated at earlier correction/prediction levels and timesteps; one can also think of the memory footprint as the discretization stencil across the different correction and prediction levels. For a \( P \)th order RIDC method, the \((P - 1)\)st correction update (i.e., solving error IVP #\((P-1)\)) requires a stencil of size \((P + 1)\), the \((P - 2)\)nd correction requires an additional \((P - 2)\) size stencil, the \((P - 2)\)nd correction requires an additional
In an RIDC method, solution values and correction terms are computed in a pipeline fashion. For example, while a processor is computing a solution to IVP (1) at \( t_{10} \), a second processor computes corrections to the numerical error at time \( t_{9} \), a third processor computes additional corrections at time \( t_{8} \), and so forth. The open circles denote solutions that are simultaneously computed. The solid circles denote stored solution values that are needed for the quadrature approximation.

The total memory footprint required for a \( P \)th order RIDC method is

\[
\left( \sum_{i=1}^{P-1} (i + 1) \right) + 1 = \frac{(P - 1)(P)}{2} + (P - 1) + 1 = \frac{P(P + 1)}{2}.
\]

In Christlieb et al. [2010b], it is shown that the ratio of timesteps taken by \( P \)th-order RIDC-Euler method, using \( K \) steps before a restart, to the number of steps taken by the forward Euler method is

\[
\gamma = 1 + \frac{(P - 1)^2}{K}.
\]

This shows that the method becomes more efficient (in terms of wall-clock time) as \( K \) increases. One does have to balance a large value of \( K \) with the possible increase in error that this may cause. A study of this balance is provided in Christlieb et al. [2010b].

Because of the staggering, start-up steps are needed to fill the memory footprint. As discussed in Christlieb et al. [2010b], one should control the start-up steps to minimize the size of the memory footprint—that is, it is more desirable to stall the predictors and lower-level correctors initially (as appropriate) until all predictors and correctors can be marched in a pipeline fashion with the minimal memory footprint. For example, Figure 2 shows the start-up routine for a fourth-order RIDC method. Initially, only the predictor advances the solution from \( t_0 \) to \( t_1 \) in step one. In steps two and three, both the predictor and first corrector are advanced to populate the memory stencil in preparation for the second corrector. In step four, only the second corrector is advanced; the predictor and first corrector are stalled because the memory stencil needed to advance the second corrector from \( t_1 \) to \( t_2 \) is the same memory stencil needed to advance the corrector from \( t_0 \) to \( t_1 \).

Although this concept is easy to grasp, the start-up algorithm looks nonintuitive at first glance. Algorithm 1 specifies the nuts and bolts of the start-up routine. The RIDC method can be run in a pipeline fashion (with the minimal memory footprint) after \( \text{startnum} - 1 \) initialization steps, where \( \text{startnum} = \min(1, \frac{p(p+1)}{2} - 1) \). For example, no initialization is required if \( p = 1 \). If \( p = 4 \), eight initialization steps are required—the RIDC method starts marching in a pipeline fashion at step nine. In the RIDC software, this start-up routine is implemented using the \text{filter} variable.
Fig. 2. Start-up routine for a fourth-order RIDC method. Observe that the predictor and lower-order correctors are occasionally stalled to ensure that a minimal memory footprint is used for the RIDC method. The fourth-order RIDC start-up takes eight steps; from step nine on, the RIDC method can be marched in a pipeline fashion.

ALGORITHM 1: RIDC Start-Up Routine

\[
\text{startnum} = \min(1, \frac{p(p+1)}{2} - 1);
\]

\[\text{for } p = 1 \text{ to } (P - 1) \text{ do}\]

\[\text{march previous levels (i.e. } 0, \ldots, (p - 1)\text{) in a pipe for one step; }\]

\[\text{march current level } (p - 1) \text{ steps; }\]

\[\text{march levels } 0, \ldots, p, \text{ in a pipe for one step; }\]

end

ALGORITHM 2: RIDC Pseudocode

\[
\text{fill memory stencil, compute startnum ;}
\]

\[\text{for } nt = \text{startnum to } NT \text{ do}\]

\[\text{for } p = 0 \text{ to } (P - 1) \text{ do in parallel}\]

\[\text{if } p = 0 \text{ then}\]

\[\text{use step to advance solution on prediction level (if } t_F \text{ not reached on prediction level);}\]

\[\text{else}\]

\[\text{use corr_fe or corr_be to advance solution on correction level } p \text{ (if } t_F \text{ not reached on correction level } p);\]

end

end

\[\text{update memory stencil ;}\]

end

The shutdown routine for the RIDC method is straightforward; each predictor and corrector only advances the solution until the final time, \( t_F \), is reached. The parallel RIDC pseudocode is summarized in Algorithm 2.

3. RIDC SOFTWARE

To utilize popular sequential integrators as described in Section 1.1, a user often specifies \( f(t, y) \), the range of integration \([0, T]\), the initial condition \( y_0 = y(0) \) (and for DASSL, the derivative \( y_0' = y'(0) \)), and integrator parameters (e.g., parameters for controlling step-size adaptivity). Although these general-purpose time integration routines are convenient and easy to use, this "black-box" approach (e.g., a user does not have to deal with the nonlinear solves arising from the backward differentiation for-
ridc methods: a family of parallel time integrators

3.1. under the hood

the RIDC software and examples are coded in C++; task parallelism is achieved using OpenMP threads to solve the predictors and the correctors in parallel. This mode of parallelism was chosen to accommodate the data movement/communication required by the RIDC algorithm when solving Equations (4) and (5). We assume that the user-defined step routine to advance the solution is a first-order sequential integrator, although with some minor modifications to the RIDC software provided, bootstrapping higher-order integrators is possible. the RIDC software can also be modified to leverage a thread-safe user-defined step routine. For example, a CUDA-accelerated step routine [Ong et al. 2012] or an MPI-parallelized step routine [Christlieb et al. 2012] can be utilized (see Section 3.3). if the step routine uses an explicit Euler integrator, the RIDC software assumes that \( u_{n+1} \) satisfies

\[
\begin{align*}
    u_{n+1} - u_n &= \Delta t f(t_n, u_n).
\end{align*}
\]

if the step routine uses an implicit Euler integrator, the RIDC software assumes that \( u_{n+1} \) satisfies

\[
\begin{align*}
    u_{n+1} - u_n &= \Delta t f(t_{n+1}, u_{n+1}).
\end{align*}
\]

The RIDC software treats this step routine as a black box, as depicted in Figure 3.

The RIDC functions solve Equations (4) and (5) by creating the necessary data structures to store copies of the solution vector described in Section 2.5 and then performing the appropriate algebraic computations on these stored solution values. First, consider the explicit Euler discretization of the error equation (4). Observe that \( u_{n+1}^{[p]} \) can be constructed by applying the user-defined step routine to \( u_{n+1}^{[p]} \) to obtain \( \tilde{v}_{n+1}^{[p]} \) and then adding \(-\Delta t f(t_n, u_{n}^{[p]}) + \int_{t_{n}}^{t_{n+1}} f(\tau, u_{\tau}^{[p]}) d\tau \) to \( \tilde{v}_{n+1}^{[p]} \) to finally obtain \( u_{n+1}^{[p]} \). The explicit RIDC wrapper is displayed in Figure 4.

A similar observation can be made about the implicit Euler discretization of the error equation (5); however, one first constructs the intermediate value \( \tilde{v}_{n}^{[p+1]} = u_{n}^{[p+1]} - \Delta t f(t_n, u_{n}^{[p]}) + \int_{t_{n}}^{t_{n+1}} f(\tau, u_{\tau}^{[p]}) d\tau \) and then applies the user-defined step function to \( \tilde{v}_{n}^{[p+1]} \). The implicit RIDC wrapper is displayed in Figure 5.

3.2. discussion

The computational overhead of RIDC methods resides mainly in the quadrature approximation and the subsequent linear combinations used to compute the corrected solutions. Provided that this computational overhead is small compared to an evaluation of the step routine, good parallel speedup is achieved. in practice, this is almost always the case for implicit RIDC methods where solutions to linear equations and/or Newton iterations are required. for explicit RIDC methods, good parallel speedup is
only observed when the step routine is sufficiently expensive, such as in the computation of self-consistent forces for an $n$-body problem [Christlieb et al. 2010b].

As mentioned in Section 2.5, the RIDC method has to store copies of the solution vector evaluated at earlier correction/prediction levels. Although this memory requirement might appear restrictive, the memory footprint for high-order single-step, multistep, or general linear methods are similar. Implicit RIDC methods also benefit from the loose coupling between the prediction and correction equations, whereas a general implicit $s$-stage implicit RK method necessitates the solution of a system of (potentially nonlinear) $sN$ equations, where $N$ are the number of differential algebraic equations. A $p$th-order RIDC method constructed using backward Euler integrators requires the solution of $p$ decoupled systems of $N$ (potentially nonlinear) algebraic equations.

3.3. Possible Generalizations

For clarity, only the simplest variant of the RIDC method (constructed using first-order Euler integrators, uniform timestepping, serial computation of the step routine) has been presented and released as part of the base software version. Here, we make some remarks on how the base version of the software can be modified by the user to accommodate several generalizations discussed in this section; indeed, the authors will release (when possible) modified versions of the software within the source repository that illustrate how to generate generalized RIDC integrators.

Step-size adaptivity for error control. In Christlieb et al. [2015], various variants of adaptive RIDC methods were presented. In the simplest variant, one uses standard error control strategies to adaptively select step sizes while solving IVP (1). These adaptively selected step sizes are used for solving the error equations (2). To build step-size adaptivity into the provided RIDC software, the following modifications will be needed: (i) modify the time-loop appropriately to allow for nonuniform steps, (ii) modify the driver file appropriately to take a user-defined tolerance (as opposed to the number
of timesteps), and (iii) recompute the integration matrix containing the quadrature weights at every timestep. The user will presumably provide an additional adapt_step function, which takes as inputs the solution at time $t$, the previous time step used, $\Delta t_{old}$, a tolerance $tol$, and returns the timestep selected, $\Delta t$, and the solution at the new timestep, $t + \Delta t$.

**Restarts.** As discussed in Christlieb et al. [2010b], the RIDC method accumulates error while running in a pipeline fashion—the most accurate solution does not propagate to the earlier prediction/correction levels. In some cases, it might be advantageous to stop the RIDC method and use the most accurate solution to “restart” the computation. This requires only a simple modification to the main RIDC loop in ridc.cpp.

**Constructing RIDC methods using higher-order integrators.** With a few modifications, it is possible to use higher-order single-step integrators within the RIDC software. The memory stencil, integration matrix, and quadrature approximations will need to be modified in ridc.cpp.

**Semi-implicit RIDC methods.** Although semi-implicit RIDC methods have been constructed and studied in Ong et al. [2012], it is generally not possible to wrap a user-defined semi-implicit step function to solve the error equation (2). Consider the IVP

$$y'(t) = f_N(t, y) + f_S(t, y),$$

where $f_S$ contains stiff terms and $f_N$ contains the nonstiff terms. A first-order user-defined step function to solve this IVP would look like

$$u_{n+1} - u_n = \Delta t f_N(t_n, u_n) + \Delta t f_S(t_{n+1}, u_{n+1}),$$

whereas the first-order IMEX discretization of the error equation (2) is

$$u^{[p+1]}_{n+1} = u^{[p+1]}_n + \Delta t \left[ f_N(t_n, u^{[p+1]}_n) + f_S(t_{n+1}, u^{[p+1]}_{n+1}) \right] - \Delta t \left[ f_N(t_n, u^{[p]}_n) + f_S(t_{n+1}, u^{[p]}_{n+1}) \right]
+ \int_{t_n}^{t_{n+1}} \left[ f_N(\tau, u^{[p]}_n) + f_S(\tau, u^{[p]}_n) \right] d\tau.$$

Although it is not obvious how to bootstrap a semi-implicit step function automatically, a user can leverage the data structures and quadrature approximations in ridc.cpp to construct a new corr_fbe function, which should look similar in structure to the users’ step function.

**Using accelerators for the step routine.** Many computing clusters feature nodes with multiple accelerators (e.g., Nvidia GPGPUs or Intel Xeon Phis). If the user wishes to provide a step routine that is accelerated using these emerging architectures, the RIDC code can be modified to leverage multiple accelerators in a computational node. Modifications that are required include adding an input variable “level” (an integer from 0 to $p - 1$, where $p$ is the desired order / number of accelerators in the system) into the step routine, a function call within the step function to specify the appropriate accelerator (e.g., cudaSetDevice for the NVIDIA GPGPUs), and a modification of ridc.cpp so that the prediction/correction level is fed into the step function, ensuring that the linear algebra is performed on the appropriate accelerator.

**Using distributed MPI for the step routine.** Although the RIDC software can be modified to allow for an MPI-distributed step routine (provided that this step routine is thread safe), we showed in Haynes and Ong [2014] that a tighter coupling of the hybrid MPI-OpenMP formulation to reduce the number of messages is necessary for performance.
4. NUMERICAL EXPERIMENT

The software includes several examples verifying that the RIDC methods attain their designed orders of accuracy. As mentioned, these examples also serve as templates for users to bootstrap their own first-order time integration methods to give a parallel-in-time approximation. Good parallel speedup is observed when the computational overhead for the RIDC methods (namely, the quadrature approximation and the linear combinations to compute the corrected solutions) is small compared to an evaluation of the step routine. Here, we present the numerical results for the Brusselator in $\mathbb{R}^1$:

$$u_t = A + u^2 v - (B + 1)u + \alpha u_{xx},$$

$$v_t = Bu - u^2 v + \alpha v_{xx},$$

with $A = 1$, $B = 3$ and $\alpha = 0.02$, initial conditions

$$u(0, x) = 1 + \sin(2\pi x), \quad v(0, x) = 3,$$

and boundary conditions

$$u(t, 0) = u(t, 1) = 1, \quad v(t, 0) = v(t, 1) = 3.$$

A central finite difference approximation is used to discretize Equation (6). The resulting nonlinear system of equations is solved using a Newton iteration. In the timing results, the Intel Math Kernel Library (MKL) is used to solve the linear system arising in each Newton iteration. The code for this example can be found in the examples/brusselator_mkl directory. Figure 6 shows a standard convergence study of error versus number of timesteps to demonstrate that the RIDC software bootstraps the first-order integrator to generate a high-order method of the desired accuracy.

In Figure 7, the wall time used to compute each RIDC method is plotted to show the “weak scaling” capability of RIDC methods. For example, the fourth-order RIDC method computes a solution using four computing cores that is three to five orders of magnitude more accurate than the first-order Euler solution in approximately the same wall-clock time. Timing results using a serial three-stage, fifth-order RADAU IIA integrator is also presented. A fifth-order RIDC method (with five computing cores)
Fig. 7. Error as a function of wall time is plotted for various RIDC methods. Here, two computing cores (set via `OMP_NUM_THREADS=2`) are used to compute the second-order RIDC method (RIDC-2), three computing cores are used to compute RIDC-3, four computing cores are used to compute RIDC-4, and five compute cores are used to compute RIDC-5. A single computing core was used to compute Radau IIA. The RIDC software computes a $p$th-order solution in approximately the same wall-clock time as an Euler solution, provided that $p$ computing cores are available. The parallel RIDC methods also provide good speedup over a serial Radau IIA integrator.

provides a solution with comparable accuracy in 10% of the wall time. The scaling studies were performed on a single computational node consisting of a dual-socket Intel E5-2670v2 chipset.

5. CONCLUSIONS
In this article, we presented the RIDC software for solving systems of initial values problems. This approach bootstraps lower-order time integrators to generate high-order approximations in approximately the same wall-clock time, resulting in a multiplicative increase in the number of compute cores utilized. The C++ framework produces a parallel-in-time solution of a system of IVPs given user-supplied code for the right-hand side of the system and the sequential code for a first-order timestep. The user-supplied timestep routine may be explicit or implicit and may make use of any auxiliary libraries that take care of the solution of the nonlinear algebraic systems that arise or the numerical linear algebra required.

REFERENCES
S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, et al. 2014. PETSc Users Manual Revision 3.7. Retrieved July 17, 2016, from http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf.

Peter N. Brown, George D. Byrne, and Alan C. Hindmarsh. 1989. VODE: A variable-coefficient ODE solver. SIAM Journal on Scientific Computing 10, 5, 1038–1051. DOI: http://dx.doi.org/10.1137/0910062

Kevin Burrage. 1993. Parallel methods for initial value problems. Applied Numerical Mathematics 11, 1–3, 5–25. DOI: http://dx.doi.org/10.1016/0168-9274(93)90037-R

Andrew Christlieb, Ronald Haynes, and Benjamin Ong. 2012. A parallel space-time algorithm. SIAM Journal on Scientific Computing 34, 5, 233–248.
David I. Ketcheson and Umair Bin Waheed. 2014. A comparison of high-order explicit Runge-Kutta, extrapolation, and deferred correction methods in serial and parallel. *Communications in Applied Mathematics and Computational Science* 9, 2, 175–200. DOI: http://dx.doi.org/10.2140/camcos.2014.9.175

Yvon Maday and Gabriel Turinici. 2002. A parareal in time procedure for the control of partial differential equations. *Comptes Rendus Mathematique* 335, 4, 387–392. DOI: http://dx.doi.org/10.1016/S1631-073X(02)02467-6

Willard Miranker and Werner Liniger. 1967. Parallel methods for the numerical integration of ordinary differential equations. *Mathematics of Computation* 21, 303–320.

Benjamin Ong, Andrew Christlieb, and Andrew Melfi. 2012. *Parallel Semi-Implicit Time Integrators*. Technical Report. Michigan State University, East Lansing, MI. http://arxiv.org/pdf/1209.4297.pdf.

Linda Petzold. 1983. A description of DASSL: A differential/algebraic system solver. In *Scientific Computing*. IMACS, New Brunswick, NJ, 65–68.

Bernhard Schmitt. 2013. Peer Methods for Ordinary Differential Equations. Retrieved July 17, 2016, from http://www.mathematik.uni-marburg.de/~schmitt/peer/.

Jacob Schroder, Robert Falgout, Tzanio Kolev, Ulrike Yang, Anders Petersson, Veselin Dobrev, Scott MacLachlan, Stephanie Friedhoff, and Ben O’Neil. 2015. XBraid: Parallel Time Integration with Multigrid. Retrieved July 17, 2016, from http://llnl.gov/casc/xbraid. (2015).

Lawrence Shampine, Mark Reichelt, and Jacek Kierzenka. 1999. Solving index-1 DAEs in MATLAB and Simulink. *SIAM Review* 41, 3, 538–552. DOI: http://dx.doi.org/10.1137/S003614459933425X

Walter Vandevender and Karen Haskell. 1982. The SLATEC mathematical subroutine library. *ACM SIGNUM Newsletter* 17, 3, 16–21.

Stefan Vandewalle and Dirk Roose. 1989. The parallel waveform relaxation multigrid method. In *Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing*. 152–156.

Rüdiger Weiner, Katja Biermann, Bernhard A. Schmitt, and Helmut Podhaisky. 2008. Explicit two-step peer methods. *Computers and Mathematics with Applications* 55, 4, 609–619. DOI: http://dx.doi.org/10.1016/j.camwa.2007.04.026

Received August 2014; revised December 2015; accepted December 2015