PARALLEL HIGH-ORDER INTEGRATORS

ANDREW J. CHRISTLIEB*, COLIN B. MACDONALD†, AND BENJAMIN W. ONG‡

Abstract. In this work we discuss a class of defect correction methods which is easily adapted to create parallel time integrators for multi-core architectures and is ideally suited for developing methods which can be order adaptive in time. The method is based on Integral Deferred Correction (IDC), which was itself motivated by Spectral Deferred Correction by Dutt, Greengard and Rokhlin (BIT-2000).

The method presented here is a revised formulation of explicit IDC, dubbed Revisionist IDC, which can achieve p^{th}-order accuracy in "wall-clock time" comparable to a single forward Euler simulation on problems where the time to evaluate the right-hand side of a system of differential equations is greater than latency costs of inter-processor communication, such as in the case of the N-body problem. The key idea is to re-write the defect correction framework so that, after initial startup costs, each correction loop can be lagged behind the previous correction loop in a manner that facilitates running the predictor and $M = p-1$ correctors in parallel on an interval which has K steps, where $K \gg p$. We prove that given an r^{th}-order Runge–Kutta method in both the prediction and M correction loops of RIDC, then the method is order $r \times (M+1)$.

The parallelization in Revisionist IDC uses a small number of cores (the number of processors used is limited by the order one wants to achieve). Using a four-core CPU, it is natural to think about fourth-order RIDC built with forward Euler, or eighth-order RIDC constructed with second-order Runge–Kutta. Numerical tests on an N-body simulation show that RIDC methods can be significantly faster than popular Runge–Kutta methods such as the classical fourth-order Runge–Kutta scheme.

In a PDE setting, one can imagine coupling RIDC time integrators with parallel spatial evaluators, thereby increasing the parallelization. The ideas behind RIDC extend to implicit and semi-implicit IDC and have high potential in this area.

Key words. Initial value problems, integral deferred correction, parallel computation, multi-core computing, N-body simulations.

AMS subject classifications. 65L05, 65Y05, 65L20

1. Introduction.

In this paper, we construct and analyze a class of novel time integrators for initial value problems (IVP), known as Revisionist Integral Deferred Correction methods (RIDC), which can be efficiently implemented with multi-core architectures. We adopt the "revisionist" terminology to highlight that (1) this is a *revision* of the standard IDC formulation, and (2) successive corrections, running in parallel but lagging in time, *revise* and improve the approximation to the solution. Although the RIDC schemes are presented in the integral deferred correction framework, this new method can equivalently be viewed as a sequence of time steps with an occasional "reset".

Integral deferred correction (IDC) methods, also known in some instances as spectral deferred correction (SDC), were first introduced by Dutt, Greengard and Rokhlin [9], and were further developed in [1, 22, 21, 18, 19, 7, 6, 29, 23, 17]. An iterative correction procedure is used to correct an estimate of the solution by solving an integral formulation of the error equation. While it has been demonstrated that IDC methods are competitive with popular high-order Runge–Kutta (RK) integrators [6], there is no systematic mechanism for spreading the workload on multi-core architectures for

*Deptartment of Mathematics, Michigan State University, East Lansing, MI 48824, USA (christli@msu.edu).

† Oxford Centre for Collaborative Applied Mathematics, Mathematical Institute, University of Oxford, 24-29 St Giles', Oxford, OX1 3LB, UK (macdonald@maths.ox.ac.uk).

‡Department of Mathematics, Michigan State University, East Lansing, MI 48824, USA (bwo@math.msu.edu).

either IDC or popular RK methods. The issue is that both IDC and RK methods are formulated in a sequential framework, in that a high-order approximation to the solution at time $t + \Delta t$ can not be computed until a high-order approximation to the solution at time t is known. This drawback of a sequential integrators is an important area to address as computing technologies are moving in the direction of multi-core architectures, and many existing algorithms will need to be modified to effectively make use of these advances.

Several approaches have been proposed to address the limitations of sequential integrators. In [31], Nievergelt proposed a framework for decomposing the time integration interval into sub-intervals, solving the initial value problem on each sub-interval concurrently, and then enforcing continuity of the solution. In [30], Miranker and Liniger discuss parallel integrators based on predictor/corrector ideas. This approach leads them to a family of integrators, which are essentially second- and third-order multi-step Adams schemes where certain steps can be computed in parallel, as well as second- and third-order RK integrators, where certain stages can be computed in parallel. A more contemporary approach are the "parareal" algorithms, where the time integration interval is once again divided into sub-intervals and a serial prediction computation is performed, followed by a parallel corrective iteration and then a serial corrective iteration. Discussion and analysis of parareal algorithms can be found in [11, 12, 24, 27, 13]. There has also been recent work in [28], using SDC methods to take successive parallel and serial corrective iterations. We note that parareal algorithms appear particularly promising for solving IVPs when a very large number of computing cores (e.g., > 32) are available.

Our proposed family of time integrators take a different approach in that the prediction and correction loops are all performed in parallel. For example, in roughly the same wall-clock time it would take to compute a first-order forward Euler approximation, our RIDC algorithm can compute a fourth-order approximation to the solution of an IVP, provided four computing cores are available for the computation. If an even higher-order integrator is required, the RIDC algorithm can compute an eighth-order approximation to the solution in roughly the same amount of wall-clock time as a forward Euler integrator, provided eight computing cores are available for the computation. Alternatively, on four cores, RIDC can achieve eighth order in the same wall clock time as a second order RK integrator, provided that second order RK integrator is embedded in IDC. A non-exhaustive list of applications where our RIDC algorithms will be advantageous includes classic N-body simulations involving charged particle interactions in plasmas [34, 8, 4], biological systems [2, 25], and gravitational bodies [3, 16]. Additionally, one can imagine applying RIDC algorithms to explicit/implicit solutions of PDEs with non-local operators such as Landau–Fokker–Planck [10] or image processing applications [14].

This paper is divided into five main sections. In §2, IDC methods are reviewed and RIDC methods are introduced. In §3, the implementation of RIDC methods in a multi-core environment is discussed. Convergence and stability of RIDC methods are presented in §4. Then, numerical benchmarks comparing IDC, RIDC, and popular Runge–Kutta methods are given in §5, followed by conclusions in §6.

2. IDC and RIDC methods.

In this section, we review IDC methods from [9], and introduce our new RIDC methods. All of these methods compute an approximate

solution to an IVP consisting of a system of ODES and initial conditions,

$$\begin{cases} y'(t) = f(t,y), & t \in [a,b], \\ y(a) = \alpha. \end{cases} \tag{2.1}$$

In this paper, we focus the bulk of our discussions on IDC and RIDC methods constructed using first-order forward Euler integrators, and note that IDC and RIDC methods can easily be constructed using higher-order implicit and explicit RK integrators if the time domain is discretized uniformly [7, 6].

IDC and RIDC methods are predictor-corrector schemes. Given an approximate solution $\eta(t)$ to the exact solution $y(t)$, the error of the approximate solution is

$$e(t) = y(t) - \eta(t). \tag{2.2}$$

If we define the residual as $\epsilon(t) = \eta'(t) - f(t,\eta(t))$, then the derivative of the error (2.2) satisfies

$$e'(t) = y'(t) - \eta'(t) = f\left(t, y(t)\right) - f\left(t, \eta(t)\right) - \epsilon(t).$$

The integral form of the error equation can then be obtained,

$$\left[e(t) + \int_0^t \epsilon(\tau)\, d\tau\right]' = f\left(t, \eta(t) + e(t)\right) - f\left(t, \eta(t)\right).$$

Substituting the definition for the residual gives

$$\left[e(t) + \eta(t) - \int_0^t f(\tau, \eta(\tau))\, d\tau\right]' = f\left(t, \eta(t) + e(t)\right) - f\left(t, \eta(t)\right). \tag{2.3}$$

The basic idea of this paper is that while a single computing core is computing a forward Euler predicted solution to IVP (2.1), other computing cores can simultaneously compute increasingly accurate corrections using equation (2.3). Each correction revises the approximate solution, improving the accuracy by, for example, one order of accuracy. Because each computation occurs simultaneously, high-order accurate results are obtained in just slightly more wall-clock time than a forward Euler computation on a single processor computer. We first describe standard IDC methods, and then introduce the revisions that result in a scheme that can be computed in parallel.

2.1. IDC methods. Suppose the time domain $[a,b]$ is uniformly discretized into N intervals, and the resulting nodes are enumerated and grouped such that there are J groups of M intervals. Specifically, if $\Delta t = \frac{b-a}{N}$, then the nodes

$$t_n = n\,\Delta t, \quad n = 0, \ldots, N,$$

are enumerated

$$t_{j,m} = (jM + m)\Delta t, \quad m = 0, \ldots, M, \quad j = 0, \ldots, J-1,$$

such that each grouping,

$$I_j = \{t_{j,0}, t_{j,1}, \ldots, t_{j,M}\}, \quad j = 0, \ldots, J-1,$$

contains $p = M + 1$ nodes. This is illustrated in Figure 2.1

IDC methods are predictor-corrector schemes that iterate completely on each group of intervals I_j sequentially, starting with $j = 0$. The idea is as follows.

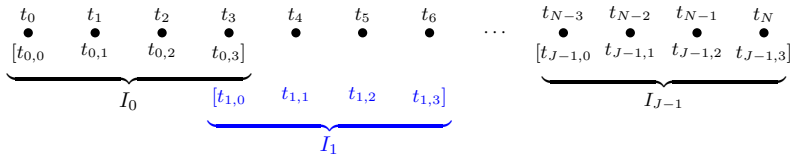


Fig. 2.1: Enumerating nodes such that each grouping contains $M = 3$ intervals.

1. Suppose that the solution is known at $t_{j,0}$. This is true for $j = 0$, where $\eta(t_{0,0}) = \alpha$.
2. Using an integrator of choice, solve the IVP $\eta' = f(t, \eta)$ for a provisional solution at all nodes contained in I_j. Denote this provisional solution as $\eta^{[0]}_{j,m}$, where $m = 0, \ldots, M$.
3. Solve the error equation (2.3), and use the error to obtain a corrected solution at all nodes contained in I_j. Denote the solution after the first correction as $\eta^{[1]}_{j,m}$, where $m = 0, \ldots, M$.
4. Repeat step 3 $(M-1)$ times, using the most recently corrected solution as an approximation to the exact solution. Denote the solution after the l^{th} correction loop as $\eta^{[l]}_{j,m}$, where $m = 0, \ldots, M$.
5. Use the accurately computed solution at $t_{j,M}$ to repeat the process for the next group of intervals, I_{j+1}. Specifically, set $\eta(t_{j+1,0}) = \eta^{[M]}_{j,M}$, $j \leftarrow j+1$ and return to step 1.

It was shown in [9] that if forward Euler integrators are used to solve for the provisional solution in step 2, and M forward Euler corrections are applied in steps 3 and 4, then the resulting algorithm is $(M+1)^{\text{st}}$-order accurate. The IDC algorithm using forward Euler integrators is written out explicitly in Algorithm 1, where line 19 shows a forward Euler discretization of equation (2.3), manipulated to give an update formula to the provisional solution $\eta^{[l-1]}_{j,m}$,

$$\begin{aligned}\eta^{[l]}_{j,m+1} &= \eta^{[l]}_{j,m} + \Delta t(f(t_{j,m}, \eta^{[l]}_{j,m}) - f(t_{j,m}, \eta^{[l-1]}_{j,m})) + \int_{t_{j,m}}^{t_{j,m+1}} f(t, \eta^{[l-1]}(t))\, dt \\ &= \eta^{[l]}_{j,m} + \Delta t(f(t_{j,m}, \eta^{[l]}_{j,m}) - f(t_{j,m}, \eta^{[l-1]}_{j,m})) + \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, \eta^{[l-1]}_{j,i}), \quad (2.4)\end{aligned}$$

where S_{mi} are quadrature weights, compactly expressed in an integration matrix whose elements are defined as integrals of Lagrange interpolating polynomials,

$$S_{mi} = \int_m^{m+1} \left(\prod_{k=0, k\neq i}^{M} \frac{t-k}{i-k} \right) dt. \quad (2.5)$$

2.1.1. IDC with Runge–Kutta integrators. It was shown in [6] that if an r^{th}-order Runge–Kutta integrator is used in the prediction and correction loops (instead of forward Euler), then the order of accuracy of the IDC method increases to order $r \times (M+1)$. For example, if a trapezoidal RK-2 discretization of (2.3) is performed instead, an update formula for the provisional solution, after algebraic

manipulations, is

$$\eta_{j,m+1}^{[l]} = \eta_{j,m}^{[l]} + \frac{K_1}{2} + \frac{K_2}{2} + \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, \eta_{j,i}^{[l-1]}),$$

where

$$K_1 = \Delta t(f(t_{j,m}, \eta_{j,m}^{[l]}) - f(t_{j,m}, \eta_{j,m}^{[l-1]})),$$
$$K_2 = \Delta t \left(f\left(t_{j,m} + \Delta t, \eta_{j,m}^{[l]} + K_1 + \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, \eta_{j,i}^{[l-1]}) \right) - f(t_{j,m} + \Delta t, \eta_{j,m+1}^{[l-1]}) \right).$$

It was also shown in [6] that using higher-order integrators to solve for the provisional and corrected solution results in IDC schemes with superior stability properties.

Input: endpoints $\{a, b\}$; initial condition α; number of intervals N; order of method p (Note that we require N to be divisible by $M = p - 1$, so that there are J groups of M intervals)

(Initialize and pre-compute integration matrix)
$\eta_{-1} \leftarrow \alpha, \quad \Delta t \leftarrow \frac{b-a}{N}, \quad M = p - 1, \quad J = \frac{N}{M}$
for *m = 0* **to** *(M-1)* **do**
  for *i = 0* **to** *M* **do**
    $S_{mi} = \int_m^{m+1} (\prod_{k=0, k\neq i}^{M} \frac{t-k}{i-k})\, dt$
  end
end
for *j = 0* **to** *(J-1)* **do**
  (Prediction Loop)
  $\eta_{j,0}^{[0]} \leftarrow \eta_{j-1}$
  for *m = 0* **to** *(M-1)* **do**
    $t_{j,m} \leftarrow (jM + m)\Delta t$
    $\eta_{j,m+1}^{[0]} \leftarrow \eta_{j,m}^{[0]} + \Delta t f(t_{j,m}, \eta_{j,m}^{[0]})$
  end
  (Correction Loops)
  for *l = 1* **to** *M* **do**
    $\eta_{j,0}^{[l]} \leftarrow \eta_{j,0}^{[l-1]}$
    for *m = 0* **to** *(M-1)* **do**
      $\eta_{j,m+1}^{[l]} \leftarrow \eta_{j,m}^{[l]} + \ldots$
        $\Delta t(f(t_{j,m}, \eta_{j,m}^{[l]}) - f(t_{j,m}, \eta_{j,m}^{[l-1]})) + \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, \eta_{j,i}^{[l-1]})$
    end
  end
  $\eta_{j+1} \leftarrow \eta_{j,M}^{[M]}$
end

Algorithm 1: IDCp-FE algorithm – a p^{th}-order IDC method constructed using forward Euler integrators for the prediction and correction loops.

2.2. Revisionist IDC methods. The proposed RIDC algorithm relaxes the requirement of iterating M correction loops completely in each group of M intervals, $I_j = \{t_{j,0}, t_{j,1}, \ldots, t_{j,M}\}$. Instead, we allow each group to contain K intervals, where $K \gg M$, but still iterate only M times, using a stencil size of $(M+1)$ to approximate the quadrature of the residual. For example, if $K = \frac{N}{2}$, then there are $J = 2$ groups and this modified grouping is illustrated in Figure 2.2.

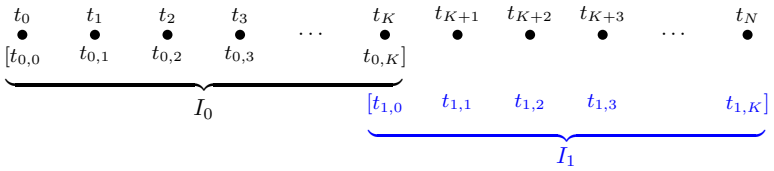


Fig. 2.2: Enumerating nodes such that each grouping contains $K+1$ nodes.

The RIDC algorithm for a p^{th}-order method (using $M = p-1$ Euler corrections) is described in Algorithm 2. Note that if $K = M$, then the RIDC method is exactly the previously formulated p^{th}-order IDC method. We note that this algorithm could also be considered as a sequence of K time steps after which a "reset" is performed, where a reset takes the most accurate solution available at $t = t_\star$ (namely $\eta^{[M]}(t_\star)$), and uses that solution as the "initial conditions" to restart the algorithm. The total number of resets performed is $J = \frac{N}{K}$.

At first glance, RIDC appears to offer little benefit over IDC. However, if there are multiple computing cores available to compute the portions of the algorithm simultaneously, then the result can potentially be computed faster. The implementation and benefit of this RIDC generalization for multi-core architectures is described in §3, and the analysis justifying this formulation is given in §4.

Lines 20 and 24 of Algorithm 2 use quadrature to approximate $\int_{t_m}^{t_{m+1}} f(t, \eta(t))\, dt$. The choice of stencils for this quadrature is not unique. In line 20, we have chosen stencils that correspond to IDC methods if $K = M$; specifically for $m < M$, we use function values at the nodes $\{t_0, t_1, \ldots, t_M\}$. The quadrature weights are consequently elements of the integration matrix S_{mi}, introduced earlier in equation (2.5). In line 24 (for $m \geq M$), we have chosen the stencil which uses the nodes $\{t_{m-M}, t_{m-(M-1)}, \ldots, t_m\}$. Since we are working with nodes that are uniformly spaced, the previously computed quadrature weights can be used to approximate the integral at each time step,

$$\int_{t_{j,m-1}}^{t_{j,m}} f(t, \eta^{[l-1]}(t))dt \approx \Delta t \sum_{i=0}^{M} S_{M-1,i} f(t_{j,m-M+i}, \eta^{[l-1]}_{j,m-M+i}).$$

2.3. RIDC methods with reduced stencils and order adaptivity. In both Algorithm 1 and Algorithm 2 for p^{th}-order IDC and RIDC methods respectively, the integral of the residual is computed by fitting an M^{th}-degree Lagrange polynomial through a data set at every correction loop, and then performing quadrature on that interpolating polynomial. However, it is not necessary to find the M^{th}-degree Lagrange interpolating polynomial at every correction loop. It suffices to find the l^{th}-degree Lagrange interpolating polynomial at the l^{th} correction loop. In §3.1.1, we will see that the reduced stencils corresponding to lower-degree polynomial interpolants

Input: endpoints $\{a, b\}$; initial condition α; number of intervals N; order of method p; number of correction loops $M = p - 1$; K intervals per group (note that we require N to be divisible by K so that there are J groups of K intervals)

(Initialize and pre-compute integration matrix)
$\eta_{-1} \leftarrow \alpha, \quad \Delta t \leftarrow \frac{b-a}{N}, \quad J = \frac{N}{K}$
for *m = 0* **to** *(M-1)* **do**
  for *i = 0* **to** *M* **do**
    $S_{mi} = \int_m^{m+1} (\prod_{k=0, k\neq i}^{M} \frac{t-k}{i-k})\, dt$
  end
end
for *j = 0* **to** *(J-1)* **do**
  (Prediction Loop)
  $\eta_{j,0}^{[0]} \leftarrow \eta_{j-1}$
  for *m = 0* **to** *(K-1)* **do**
    $t_{j,m} \leftarrow (jK + m)\Delta t$
    $\eta_{j,m+1}^{[0]} \leftarrow \eta_{j,m}^{[0]} + \Delta t f(t_{j,m}, \eta_{j,m}^{[0]})$
  end
  (Correction Loops)
  for *l = 1* **to** *M* **do**
    $\eta_{j,0}^{[l]} \leftarrow \eta_{j,0}^{[l-1]}$
    for *m = 0* **to** *(M-1)* **do**
      $\eta_{j,m+1}^{[l]} \leftarrow \eta_{j,m}^{[l]} + \ldots$
        $\Delta t(f(t_{j,m}, \eta_{j,m}^{[l]}) - f(t_{j,m}, \eta_{j,m}^{[l-1]})) + \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, \eta_{j,i}^{[l-1]})$
    end
    for *m = M* **to** *(K-1)* **do**
      $\eta_{j,m+1}^{[l]} \leftarrow \eta_{j,m}^{[l]} + \Delta t(f(t_{j,m}, \eta_{j,m}^{[l]}) - f(t_{j,m}, \eta_{j,m}^{[l-1]})) + \ldots$
        $\Delta t \sum_{i=0}^{M} S_{M-1,i} f(t_{j,m-M+i}, \eta_{j,m-M+i}^{[l-1]})$
    end
  end
  $\eta_{j+1} \leftarrow \eta_{j,K}^{[M]}$
end

Algorithm 2: RIDC(p, K)-FE algorithm – a p^{th}-order RIDC method constructed using K subintervals, and forward Euler integrators for the prediction and $M = p - 1$ correction loops.

lead to faster start up times. Also lower-degree interpolating polynomials may be less oscillatory [32]. Algorithm 3 describes the RIDC method with reduced stencils. Note that the integration matrices can still be precomputed.

If one is willing to store the solution for an entire group of intervals, $\eta_{j,m}^{[l]}$ $0 \leq m \leq K$, the error at $t_{j,M}$ can be estimated to determine if an additional correction loop is required to guarantee that the error is below some tolerance. Note that this form of p-adaptivity (i.e., adaptivity in terms of order) was not previously obvious for p^{th}-order IDC methods where there was little motivation for taking fewer than M

correction loops, and no obvious way to make use of previously computed values to generate an $(p+1)^{st}$-order method.

Input: endpoints $\{a, b\}$; initial condition α; number of intervals N; order of method p; number of correction loops $M = p - 1$; K intervals per group (note that we require N to be divisible by K so that there are J groups of K intervals)

(Initialize and pre-compute integration matrices)
$\eta_{-1} \leftarrow \alpha, \quad \Delta t \leftarrow \frac{b-a}{N}, \quad J = \frac{N}{K}$
for l = *1* **to** M **do**
  for m = *0* **to** *(l-1)* **do**
    for i = *0* **to** l **do**
      $S^l_{mi} = \int_m^{m+1} (\prod_{k=0,k\neq i}^{l} \frac{t-k}{i-k})\, dt$
    end
  end
end
for j = *0* **to** *(J-1)* **do**
  (Prediction Loop)
  $\eta^{[0]}_{j,0} \leftarrow \eta_{j-1}$
  for m = *0* **to** *(K-1)* **do**
    $t_{j,m} \leftarrow (jK + m)\Delta t$
    $\eta^{[0]}_{j,m+1} \leftarrow \eta^{[0]}_{j,m} + \Delta t f(t_{j,m}, \eta^{[0]}_{j,m})$
  end
  (Correction Loops)
  for l = *1* **to** M **do**
    $\eta^{[l]}_{j,0} \leftarrow \eta^{[l-1]}_{j,0}$
    for m = *0* **to** *(l-1)* **do**
      $\eta^{[l]}_{j,m+1} \leftarrow$
      $\eta^{[l]}_{j,m} + \Delta t(f(t_{j,m}, \eta^{[l]}_{j,m}) - f(t_{j,m}, \eta^{[l-1]}_{j,m})) + \Delta t \sum_{i=0}^{l} S^l_{mi} f(t_{j,i}, \eta^{[l-1]}_{j,i})$
    end
    for $m = l$ **to** *(K-1)* **do**
      $\eta^{[l]}_{j,m+1} \leftarrow \eta^{[l]}_{j,m} + \Delta t(f(t_{j,m}, \eta^{[l]}_{j,m}) - f(t_{j,m}, \eta^{[l-1]}_{j,m})) + \ldots$
      $\Delta t \sum_{i=0}^{l} S^l_{M-1,i} f(t_{j,m-M+i}, \eta^{[l-1]}_{j,m-M+i})$
    end
  end
  $\eta_{j+1} \leftarrow \eta^{[M]}_{j,K}$
end

Algorithm 3: RIDC(p, K)-FE algorithm with reduced stencils – a p^{th}-order RIDC method constructed using K subintervals, forward Euler integrators for the prediction and $M = p - 1$ correction loops, and reduced stencils.

3. Multi-core / Multi-cpu implementation. RIDC methods can be efficiently computed in a multi-core, multi-cpu environment (e.g., an shared memory machine), or graphics processing unit (GPU) environment, where information can be conveyed rapidly from one computing core to another. In this section, we outline the

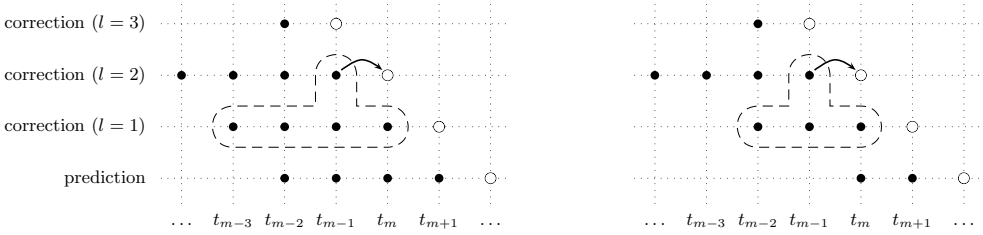
strategy for using multiple computing cores, and discuss implementations for minimizing memory usage. The multi-core implementation of RIDC methods excels when the function $f(t, y)$ in equation (2.1) is expensive to compute. This is a realistic scenario for many problems of interest.

3.1. Using multiple computing cores. We outline the strategy for using p computing cores to solve an p^{th}-order RIDC method given in Algorithm 2. These ideas also extend to RIDC methods with reduced stencils given in Algorithm 3.

For each group of intervals $I_j = \{t_{j,0}, t_{j,1}, \ldots, t_{j,K}\}$, the strategy is as follows. We drop the subscript j, with the understanding that this RIDC method is described for one group of intervals, i.e., we set $t_m := t_{j,m}$ and $\eta_m = \eta_{j,m}$.

- Start one computing core on the prediction loop to compute $\eta_m^{[0]}, m = 1, \ldots, K$ using the forward Euler scheme.
- Use a second core to compute the first correction loop, i.e., $\eta_m^{[1]}, m = 1, \ldots, K$, once sufficient information is available from the prediction loop. Specifically, the second core computes $\eta_m^{[1]}$ after $\eta_z^{[0]}$, $\eta_{z+1}^{[0]}$, $\ldots$, $\eta_{z+M}^{[0]}$ have been computed by the first computing core, where $z = \max(m - M, 0)$.
- Use a third core to compute the second correction loop, i.e., $\eta_m^{[2]}, m = 1, \ldots, K$, once sufficient information is available from the first correction loop.
- Use a fourth core to compute the third correction loop, once sufficient information is available from the second correction loop, and so on.

Figure 3.1 shows a schematic sketch of the information needed to compute $\eta_m^{[2]}$ for a fourth-order RIDC method. Some care must be taken to ensure that the l^{th} processor does not overtake the $(l-1)^{\text{st}}$ processor. This can be easily implemented using the `multiprocessing` module in Python [33]. For optimal use of memory, the l^{th} processor should not get too far ahead of $(l+1)^{\text{st}}$ processor: that way, older values of $\eta^{[l]}$ and $f(\eta^{[l]})$ which are no longer needed can be discarded.



(a) RIDC with full stencils (b) RIDC with reduced stencils

Fig. 3.1: The stencil needed to compute $\eta_m^{[2]}$ is shown in the dashed region for a fourth-order RIDC method. Additionally, this figure shows the minimum memory "foot print" if special care is taken to minimize the amount of data that has to be stored. Specifically, if the processors are not allowed to get too far ahead of each other, then the minimum data values that have to be stored to compute the open circles are shown as black dots. Central to our algorithm, is the realization that the prediction and each of the three corrections can be computed simultaneously and independently. After all four computations have completed, the oldest value at each level can be discarded and the process repeated.

3.1.1. Analysis of running time. The computing cores in the RIDC algorithm cannot start simultaneously: each must wait for the previous to compute sufficient η values. This is illustrated in Figure 3.2. The p^{th} processor (running the M^{th} correction) must initially wait for M^2 steps. This wait happens at the beginning of each group of intervals. There are $J = \frac{N}{K}$ groups of intervals overall. Thus, in theory,

(a) RIDC with full stencils (b) RIDC with reduced stencils

Fig. 3.2: Starting the RIDC4 algorithm. Each number indicates the step in which the corresponding value of $f(\eta)$ is computed. Here $M = 3$ and in (a) the third correction waits $M^2 = 9$ steps. In (b) the third correction waits for $\frac{M(M+1)}{2} = 6$ steps.

a p-core implementation of a p^{th} order RIDC method constructed using J groups of K intervals and forward Euler integrators in the prediction and correction loops (denoted RIDC(p, K)) can be computed in $N + JM^2$ steps. This can be compared to the forward Euler method running on a single processor which takes N steps. We can consider the ratio as $\gamma = \frac{N+JM^2}{N} = 1 + \frac{M^2}{K}$.

For example, consider the scenario where one discretizes a time domain into 60 intervals, and applies a forward Euler time integrator. If that simulation takes one minute on a single processor, then an RIDC(4,60) in theory will generate a fourth-order solution in 1 minute and 9 seconds if four computing cores are available for the computation. Here, $\gamma = 1.15$ and one would expect the RIDC simulation to take 15% longer than that of forward Euler. The ratio γ decreases if $K \gg M$. For example, a four core implementation of RIDC(4,600) in theory has a 1.5% increase in clock time over a single core forward Euler time integrator with 600 intervals. Practical implementation details, such as communication delays, shared memory overhead or other interprocess communication issues, would be expected to increase the wall clock time for the RIDC method. Actual timing tests are provided in §5 for a system of interacting particles.

For the RIDC method with the reduced stencils, the start up delay for each group of intervals is roughly halved: the ratio γ is reduced to $\gamma = 1 + \frac{1}{K}\frac{M(M+1)}{2}$. An obvious question to pose is how to choose K. As K increases, the ratio between the computational and idle processor time increases. However, this apparent gain in efficiency may be offset by an accumulation of error. A careful numerical study is performed in Example 5.2.

3.2. Other RIDC algorithms. As discussed in §2.3, for truly p-adaptive (i.e., variable order) schemes, the error $e_m^{[l]}$ of the solution after the l^{th} correction could be estimated before deciding whether to perform an additional correction. In terms of implementation, this may require that the data for the entire l^{th} correction level be stored. The practicality and performance of p-adaptive schemes is not explored in this paper.

We have described RIDC using forward Euler integrators. However, if the number of computing cores available is less than the order of the method desired, one could consider embedding higher-order integrators in the prediction and/or correction loops, as was done for IDC in [7, 6] and described in §2.1.1.

4. Convergence and stability of RIDC methods. We first discuss convergence of RIDC methods, followed by the stability regions of RIDC methods, and the impact of the number of intervals on the error.

4.1. Convergence of RIDC methods. The analysis in [7] (and summarized in [6]), proving convergence under mild conditions for IDC methods, extends simply to these RIDC methods.

THEOREM 4.1. *Let $f(t,y)$ and $y(t)$ in IVP (2.1) be sufficiently smooth. Then, the local truncation error for an RIDC method constructed using $K > p$ uniformly distributed nodes $\{t_k = kh, k = 0, \ldots, K\}$, an Euler prediction and $M = p-1$ Euler correction loops is $\mathcal{O}(h^{p+1})$.*

Proof. The proof follows from the analysis in [7]. We simply replace $\{M \leftarrow K, k_l \leftarrow M\}$ in Theorem 4.1 of [7], and the proof follows. We refer the reader to [7] for the actual proofs. ∎

THEOREM 4.2. *Let $f(t,y)$ and $y(t)$ in IVP (2.1) be sufficiently smooth. Then, the local truncation error for an RIDC method constructed using $K > M+1$ uniformly distributed nodes $\{t_k = kh, k = 0, \ldots, K\}$, an ${r_0}^{th}$-order Runge–Kutta method in the prediction loop, and $(r_1, r_2, \cdots, r_M)^{th}$-order RK methods in the correction loops, is $\mathcal{O}(h^{s_M+1})$, where $s_M = \sum_{j=0}^{M} r_j$.*

Proof. The proof follows from the analysis in [7]. We simply replace $\{M \leftarrow K, k_l \leftarrow M\}$ in Theorem 5.3 of [7], and the proof follows. We refer the reader to [7] for the actual proofs. ∎

4.2. Stability of RIDC methods. DEFINITION 4.3. *The amplification factor for a numerical method, $Am(\lambda\Delta t)$, can be interpreted as the numerical solution to Dalhquist's test equation,*

$$y'(t) = \lambda y(t), \quad y(0) = 1, \tag{4.1}$$

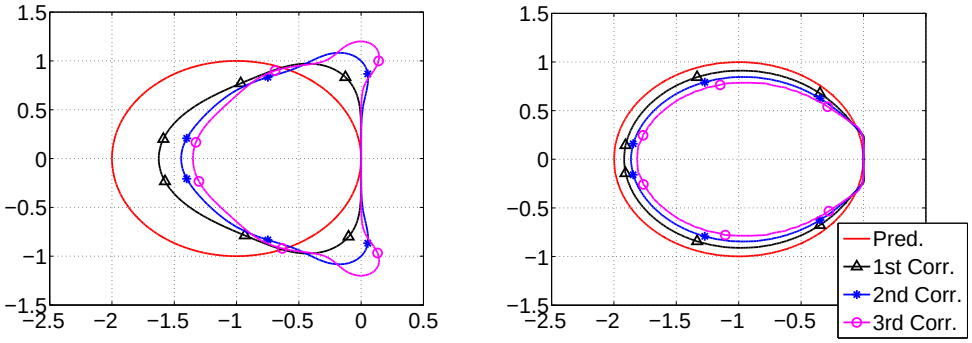
after a time step of size Δt for $\lambda \in \mathbb{C}$, i.e., $Am(\lambda\Delta t) = y(\Delta t)$.

DEFINITION 4.4. *The stability region, S, for a numerical method, is the subset of the complex plane $\mathbb{C}$, consisting of all $\lambda\Delta t$ such that $Am(\lambda\Delta t) \leq 1$,*

$$S = \{\lambda\Delta t : Am(\lambda\Delta t) \leq 1\}.$$

In Figure 4.1, we plot the stability regions for RIDC(4,4) and RIDC(4,40) after the prediction and correction loops to show how the corrective loops alter the stability regions. Note that to generate this plot, we take $N = K = 4$ and $N = K = 40$, respectively, in Algorithm 2, and scale the resulting stability regions by the number of intervals in the group, K. Correspondingly, one observes a circle of radius one for the stability region after the predictive loop, which is consistent with the forward Euler integrator used. Interestingly, RIDC(4,4) (which corresponds to the standard fourth-order IDC method) encompasses an increasing amount of the imaginary axis as the number of correction loops is increased, whereas the imaginary inclusion of the RIDC(4,40) scheme is reduced after the second corrective loop.

The change in the shape of the stability region after the third correction loop for RIDC(4,4) and RIDC(4,40) is not particularly surprising, since one might expect the



(a) RIDC(4,4), which corresponds to the standard fourth order IDC method.

(b) RIDC(4,40)

Fig. 4.1: Stability regions for (a) RIDC(4,4) and (b) RIDC(4,40) are shown for the prediction (simply that of FE) and each of the three corrections. The third correction shows the actual stability of the method. While the radius of the largest disk that can be contained in RIDC(4,40) stability region after third corrective loop is larger than that of RIDC(4,4), the loss of the imaginary inclusion is disappointing.

stability region to approach that of forward Euler as K increases. Correspondingly, we plot the stability regions for several fourth-order RIDC schemes, i.e., the stability regions for RIDC(4,K) for various K in Figure 4.2. The main observations that one should make are, for a fixed order (i) the amount of imaginary inclusion decreases as the number of intervals K increases, and (ii) the stability region approaches that of the forward Euler method as K increases.

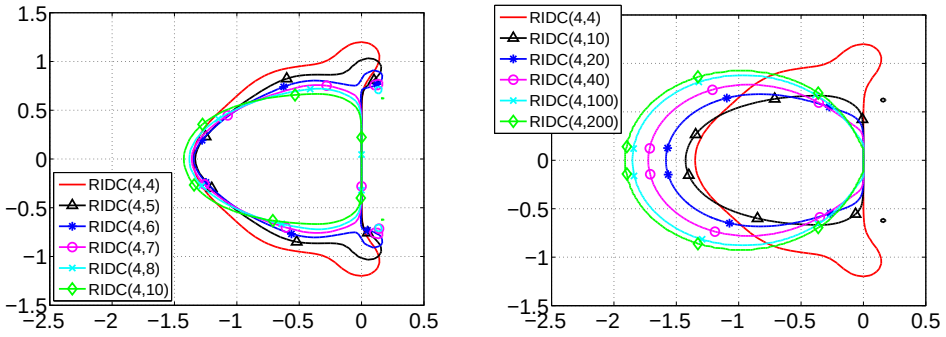
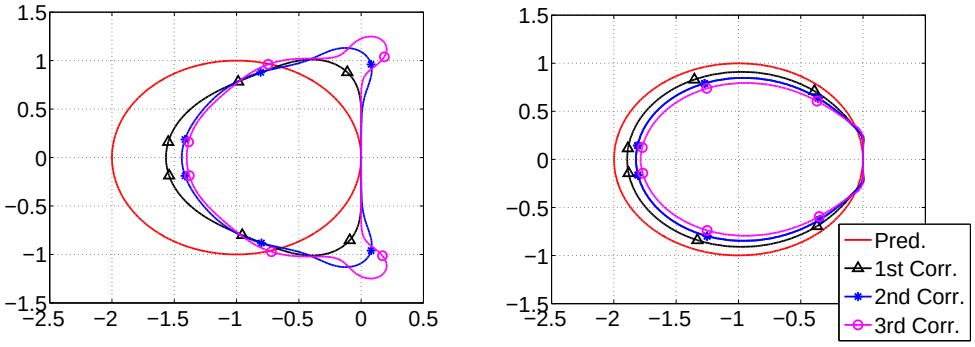


Fig. 4.2: Stability regions for RIDC methods for various K's. Notice that the size of the imaginary axis included decreases as the number of intervals K is increased. However, the stability region approaches that of the forward Euler method as K increases.

For completeness, we also present the plots of the RIDC methods with reduced stencils in Figure 4.3. There is no discernible difference between the stability regions

of RIDC methods with full stencils shown in Figure 4.1 and the RIDC methods using the reduced stencils in Figure 4.3.



(a) RIDC(4,4) w/ reduced stencils (b) RIDC(4,40) w/ reduced stencils

Fig. 4.3: Stability regions for (a) RIDC(4,4) with reduced stencils and (b) RIDC(4,40) with reduced stencils. No significant differences are observed between the stability regions for RIDC methods and RIDC methods using reduced stencils.

4.3. Choice of K, the number of intervals in each group. As mentioned in §3, maximizing the ratio $\frac{K}{M}$ for multi-core systems minimizes the amount of idle processor time. However as the ratio $\frac{K}{M}$ is increased, the overall accuracy of the solution is expected to decrease. In §5 example 5.2, we explore the impact of the number of intervals K for a test equation, noting that the optimal choice of K is likely to be problem dependent.

5. Numerical benchmarks. EXAMPLE 5.1. *This first example validates the order of accuracy of the constructed RIDC schemes. We consider the initial value problem*

$$\begin{cases} y'(t) = 4t\sqrt{y}, \quad t \in [0,5], \\ y(0) = 1. \end{cases} \tag{5.1}$$

which has the analytic solution $y(t) = (1+t^2)^2$.

Table 5.1 shows the error and rate of convergence for the RIDC schemes, where RIDC$(p, 40)$ denotes the p^{th}-order RIDC scheme described in Algorithm 2, constructed using $K = 40$ intervals per group. The numerical solution was compared to the analytic solution. In Table 5.2, a convergence test is shown for the RIDC schemes using reduced stencils.

EXAMPLE 5.2. *In this example, we consider the nonlinear ODE system presented in [1],*

$$\begin{cases} y_1' = -y_2 + y_1(1 - y_1^2 - y_2^2), \\ y_2' = y_1 + 3y_2(1 - y_1^2 - y_2^2), \\ y(0) = (1,0)^T, \quad t \in [0,10], \end{cases} \tag{5.2}$$

which has the analytic solution $y(t) = (\cos t, \sin t)^T$. We use this example to study the choice of K, the number of intervals in each grouping. This particular problem

–	RIDC(2,40)		RIDC(3,40)		RIDC(4,40)	
N	error	order	error	order	error	order
40	6.06e-03	–	4.77e-04	–	4.30e-05	–
80	1.30e-03	2.22	4.83e-05	3.30	2.26e-06	4.25
120	5.21e-04	2.26	1.19e-05	3.46	3.64e-07	4.51
160	2.73e-04	2.25	4.36e-06	3.49	9.80e-08	4.56
200	1.65e-04	2.24	2.01e-06	3.48	3.57e-08	4.53

–	RIDC(5,40)		RIDC(6,40)	
N	error	order	error	order
40	3.31e-06	–	2.55e-07	–
80	8.82e-08	5.23	3.49e-09	6.20
120	8.92e-09	5.65	2.25e-10	6.76
160	1.70e-09	5.76	3.07e-11	6.93
200	4.75e-10	5.71	6.83e-12	6.73

Table 5.1: RIDC methods exhibit their expected rate of convergence on Example 5.1. Here the error is the absolute error: the difference between the numerical and analytic solution.

–	RIDC(2,40)		RIDC(3,40)		RIDC(4,40)	
N	error	order	error	order	error	order
40	6.06e-03	–	3.44e-04	–	2.25e-05	–
80	1.30e-03	2.22	3.12e-05	3.44	9.82e-07	4.52
120	5.21e-04	2.26	7.18e-06	3.66	1.35e-07	4.89
160	2.73e-04	2.25	2.45e-06	3.74	3.22e-08	4.99
200	1.65e-04	2.24	1.06e-06	3.76	1.07e-08	4.95

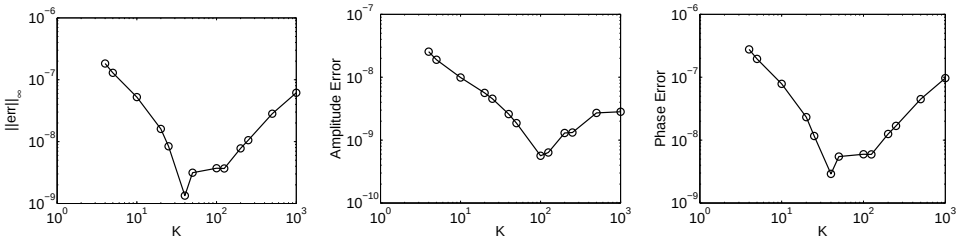
–	RIDC(5,40)		RIDC(6,40)	
N	error	order	error	order
40	1.49e-06	–	9.91e-08	–
80	3.11e-08	5.58	9.88e-10	6.65
120	2.59e-09	6.13	4.92e-11	7.40
160	4.31e-10	6.23	5.95e-12	7.34
200	1.11e-10	6.06	1.49e-12	6.22

Table 5.2: RIDC methods with reduced stencils still exhibit their expected rate of convergence on Example 5.1. Interestingly, the errors are smaller than that obtained by RIDC methods with full stencils.

was chosen because additional properties of the system (e.g., the squared amplitude error, $y_1^2 + y_2^2$, and the corresponding phase error) can be studied in addition to the error of the actual components.

In this numerical study, we fix the number of intervals at $N = 1000$, and show the effect of the interval groupings in Figure 5.1 for RIDC(4,K). Three different measures of error are shown and in all three, it appears advantageous to select K larger than $M = 4$. There is a minimum in error for K between 40 and 100, after which the error

increases again. We expect this minimum to be problem dependent.



(a) Absolute error vs K (b) Squared amplitude error vs K (c) Phase error vs K

Fig. 5.1: Three different measures of error in Example 5.2 as a function of K. Here the total number of intervals is held fixed at $N = 1000$ and RIDC$(4, K)$ is used. Note in each measure of error, the minimum occurs for $K \gg 4$.

EXAMPLE 5.3. *Our final example is the following one-dimensional N-body problem. Initially we have N_+ ions uniformly spaced on the interval $[0, 1]$, and N_- electrons also uniformly spaced on the interval $[0, 1]$, so that the total number of particles is $N_T = N_+ + N_-$. The charge of each ion and electron is set to $q_+ = \frac{1}{N_+}$ and $q_- = -\frac{1}{N_-}$ respectively so that the overall system is charge neutral. The mass of each ion and electron is set to $m_+ = \frac{1000}{N_+}$ and $m_- = \frac{1}{N_-}$, which is a physically reasonable mass ratio to work with. Denote the ion locations and velocities as $\{x_i^+, v_i^+\}_{i=1}^{N^+}$ and the electron locations and velocities as $\{x_i^-, v_i^-\}_{i=1}^{N^-}$. Then, the equations of motion are given by*

$$\begin{aligned}
\begin{bmatrix} x_i^+ \\ v_i^+ \end{bmatrix}_t &= \begin{bmatrix} v_i^+ \\ \frac{q_+}{m_+}\left(\sum_{j=1}^{N_+} \frac{q_+(x_i^+ - x_j^+)}{\sqrt{(x_i^+ - x_j^+)^2 + d^2}} + \sum_{j=1}^{N_-} \frac{q_-(x_i^+ - x_j^-)}{\sqrt{(x_i^+ - x_j^-)^2 + d^2}}\right) \end{bmatrix}, \; i = 1, \ldots, N_+, \\
\begin{bmatrix} x_i^- \\ v_i^- \end{bmatrix}_t &= \begin{bmatrix} v_i^- \\ \frac{q_-}{m_+}\left(\sum_{j=1}^{N_+} \frac{q_+(x_i^- - x_j^+)}{\sqrt{(x_i^- - x_j^+)^2 + d^2}} + \sum_{j=1}^{N_-} \frac{q_-(x_i^- - x_j^-)}{\sqrt{(x_i^- - x_j^-)^2 + d^2}}\right) \end{bmatrix}, \; i = 1, \ldots, N_-,
\end{aligned} \tag{5.3}$$

where $d > 0$ is a regularization constant. The initial conditions chosen are

$$\begin{aligned}
x_i^+(0) &= \frac{i - 0.5}{N_+}, \quad v_i^+(0) = 0, \quad i = 1, \ldots, N_+, \\
x_i^-(0) &= \frac{i - 0.5}{N_-}, \quad v_i^-(0) = \sin\big(6\pi x_i^-(0)\big), \quad i = 1, \ldots, N_-.
\end{aligned}$$

The time to evaluate the right-hand-ride scales like the square of the number of particles.

We implement the RIDC algorithm on a multi-core architecture, and demonstrate the speedup obtained with such an implementation. Our implementation uses the `multiprocessing` module in Python [33]. Computations and timings are performed on a 16-core machine: a Dell Poweredge R900 with four quad-core Xeon X7350 CPUs running at 2.93 GHz sharing 32 GiB of memory.

			time		ratio γ		
Method	N	K	theoretical	actual	theoretical	actual	error
FE	320	n/a		50.30s			8.63e00
RK4	320	n/a	201.20s	201.46s	4	4.01	9.32e-06
RIDC(4,N)	320	320	51.24s	59.14s	1.02	1.18	1.30e-03
RIDC(4,$N/2$)	320	160	52.19s	60.77s	1.04	1.21	3.66e-04
RIDC(4,$N/4$)	320	80	54.07s	62.73s	1.08	1.25	2.04e-04
RIDC(4,$N/8$)	320	40	57.84s	68.96s	1.15	1.37	5.81e-04
RK8	320	n/a	653.90s	666.74s	13	13.26	7.15e-11
RIDC8	320	320	55.96s	66.66s	1.11	1.33	4.19e-06

Table 5.3: Wall-clock timings on the particle problem. Fourth- and eighth- order RIDC schemes utilize four and eight cores respectively, FE and RK schemes use only one core. The "theoretical time" is based on comparing to the actual time of the forward Euler method. For the Runge–Kutta schemes it is the number of stages (here 4 and 13 respectively) times the forward Euler time. For the RIDC schemes (with reduced stencils), it is based on the analysis in §3.1.1.

Our simulations use 400 particles ($N_+ = N_- = 200$) and $d = 0.05$. Tables 5.3 and 5.4 compare the forward Euler integrator, fourth- and eighth-order RIDC (using four and eight cores respectively), and two Runge–Kutta schemes (using one core only). Here RK4 is the popular fourth-order Runge–Kutta scheme and RK8 is an eighth-order, 13-stage Runge–Kutta [26]. For a fixed number of intervals $N = 320$, Table 5.3 shows that the Runge–Kutta schemes take significantly longer than forward Euler, whereas the RIDC methods take approximately the same wall-clock time as forward Euler. We note the theoretical wall-clock time based on the analysis in §3.1.1 does not include overhead due to inter-process communication, latency or other concerns. The actual observed ratios γ are close to, although larger than, their theoretical values. It may be possible to improve our Python implementation. For fourth-order RIDC, we observe the expected increase in γ for smaller values of K (fewer intervals in each group). We observe a dependence of error on K which is consistent with the results of Example 5.2.

The errors shown in Table 5.3 are potentially misleading in the sense that the errors for RK4 and RK8 are superior to those of the RIDC methods. However, the actual computation time is also much larger. A direct comparison between the methods is in Table 5.4 where N is chosen such that each method computes the solution at $T = 10$ in roughly the same fixed amount of wall-clock time. Here it is clear that RIDC8 computed with eight cores is vastly superior to the other schemes and in particular the eighth-order Runge–Kutta method.

Figures 5.2 and 5.3 show studies of the error in Example 5.3 for RIDC2 through RIDC8 for a different numbers of intervals N. Table 5.4 can be seen as a vertical slice through Figures 5.2 (right) and Figure 5.3 (right). The left plot of Figure 5.2 shows the rate of convergence of second-, third- and fourth-order RIDC methods for the error as a function of the number of intervals. We note that RIDC4 has a larger error *per step* than the popular RK4 Runge–Kutta scheme by about one order of magnitude. However, the right plot of Figure 5.2 shows the error as a function of the wall-clock time, the appropriate measure of effectiveness for these multi-core simulations. Using four cores, we observe a roughly twofold speedup in RIDC4 over the popular RK4

Method	N	time	error
FE	762	119.9s	3.60e00
RK4	191	121.4s	8.93e-05
RIDC(4,N)	656	120.3s	6.50e-05
RIDC(4,$N/2$)	650	119.4s	1.75e-05
RIDC(4,$N/4$)	636	119.6s	1.05e-05
RK8	58	120.7s	1.21e-04
RIDC8	600	119.5s	1.68e-08

Table 5.4: For a fixed amount of wall-clock time, the error at $T = 10$ resulting from the various integrators is shown. These results are for the particle problem Example 5.3 and the error is the relative error in electron position. Note RIDC8 has lower error than the other schemes.

integrator.

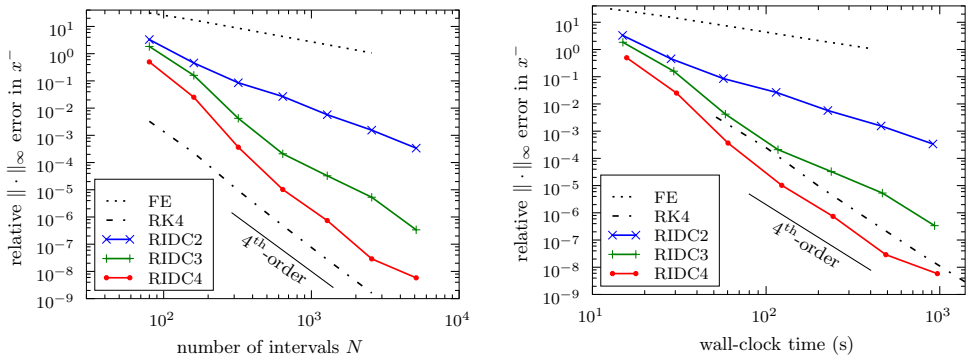


Fig. 5.2: Convergence study for the particle problem (Example 5.3) using RIDC2, RIDC3 and RIDC4 (using two, three, and four cores respectively) with $K = \min(N, 160)$ and reduced stencils. A convergence study (left) shows the methods achieve their design orders. In particular, note that RK4 has significantly better error *per step*. However, examining the wall-clock time (right), reveals that RIDC4 can be more efficient than RK4.

Figure 5.3 (left), we note that the higher-order RIDC5, RIDC6, RIDC7 and RIDC8 schemes also achieve their design orders. In the wall-clock timings in Figure 5.3 (right), we see that RIDC8 running on eight cores significantly outperforms RK4 and RK8, an eighth-order Runge–Kutta scheme [26]. In particular, RIDC8 can produce an error of 10^{-8} roughly 10 times faster than RK4 and roughly four times faster than RK8. Put another way, given the same fixed amount of wall-clock time, RIDC8 can produce an error roughly four orders of magnitude smaller than the eighth-order Runge–Kutta scheme.

6. Conclusion. In this work we discussed a class of defect correction methods which is easily adapted to create parallel time integrators for multi-core architectures. The class is also well-suited for developing methods which can be order-adaptive in

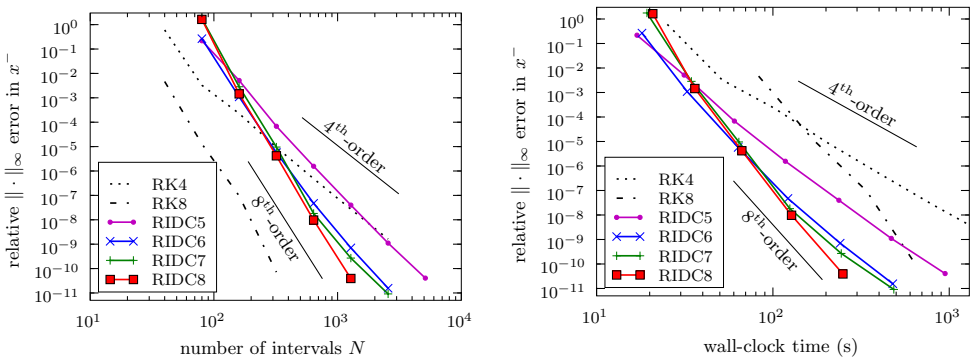


Fig. 5.3: Convergence study for the particle problem (Example 5.2) using high-order RIDC5, RIDC6, RIDC7 and RIDC8 (using five, six, seven, and eight cores respectively) with $K = N$ and reduced stencils. A convergence study (left) shows the methods achieve their design orders up to order 8. The wall-clock time (right) shows that the RIDC significantly outperform high-order Runge-Kutta schemes.

time. The methods presented are a revised formulation of explicit integral deferred correction, dubbed revisionist IDC, which can achieve p^{th}-order accuracy in wall-clock time comparable to a single forward Euler integration.

The key idea is to re-write the defect correction framework so that, after initial startup costs, each correction loop can be lagged behind the previous correction loop in a manner that facilitates running the $M = p-1$ correctors and predictor in parallel on an interval which has K steps, where $K \gg p$. In addition, we proved that given an r^{th}-order Runge–Kutta method in both the prediction and correction loops of RIDC, then the method is order $r \times (M+1)$ after M correction loops.

Numerical experiments were presented to validate our parallel integrators. A four-core implementation of fourth-order RIDC4 gives a speedup of roughly two over the popular fourth-order Runge–Kutta integrator for an N-body problem. On the same problem, eighth-order RIDC8 running on eight cores produces an error roughly four orders of magnitude smaller than an eighth-order Runge–Kutta method in the same wall-clock time.

The ideas behind RIDC extend to implicit and semi-implicit IDC [5] and have high potential in this area. They also extend to other defect correction methods (e.g., [1]). The authors are presently exploring strong stability preserving [20, 15] and other nonlinear stability properties of these RIDC schemes.

An interesting topic yet to be fully explored is the issue of adaptivity. With the advent of modern day computing architectures such as the Intel Larrabee chip and the Nvidia Tesla chip, it is not far fetched to imagine that a user might have access to a large number of computing cores. The question arises thus, how do order-adaptive RIDC methods (where, if the residual after the p^{th} correction loop is too large, an additional correction iteration is locally performed) compare with the usual paradigm of locally refining the time step? In the order adaptive paradigm, efficient integrators (such as RK4) can be embedded within the correction framework whereas in the usual paradigm of locally refining the time step, the resulting non-uniform stencil may constrain the RIDC method to incorporating only first order correctors. The

authors are researching adaptive RIDC schemes.

Acknowledgements. This work was supported by IPAM, AFOSR contract FA9550-07-01-0092, NSF grant numbers CCF-0321917 and DMS-0811175, and a fellowship from NSERC Canada. The authors would also like to acknowledge discussions with Raymond J. Spiteri that influenced this work.

REFERENCES

[1] W. Auzinger, H. Hofstätter, W. Kreuzer, and E. Weinmüller, *Modified defect correction algorithms for ODEs part I: General theory*, Numer. Algorithms, 36 (2004), pp. 135–156.

[2] N. Baker, *Improving implicit solvent simulations: a Poisson-centric view*, Current opinion in structural biology, 15 (2005), pp. 137–143.

[3] J. Barnes and P. Hut, *A hierarchical O(N log N) force-calculation algorithm*, Nature, 324 (1986), p. 4.

[4] M. Bonitz, D. Block, O. Arp, V. Golubnychiy, H. Baumgartner, P. Ludwig, A. Piel, and A. Filinov, *Structural properties of screened Coulomb balls*, Phys. Plasmas Phys Rev Lett, 96 (2005), p. 075001.

[5] A. Christlieb, M. Morton, B. Ong, and J.-M. Qiu, *Semi-implicit integral deferred correction constructed with high order additive Runge-Kutta integrators*, submitted.

[6] A. Christlieb, B. Ong, and J.-M. Qiu, *Comments on high order integrators embedded within integral deferred correction methods*, Comm. Appl. Math. Comput. Sci., 4 (2009), pp. 27–56.

[7] ———, *Integral deferred correction methods constructed with high order Runge-Kutta integrators*, Math. Comp., to appear (2009).

[8] A. J. Christlieb, R. Krasny, J. P. Verboncouer, J. W. Emhoff, and I. D. Boyd, *Grid-free plasma simulation techniques*, IEEE Trans. on Plasma Science, 34 (2006), pp. 149–165.

[9] A. Dutt, L. Greengard, and V. Rokhlin, *Spectral deferred correction methods for ordinary differential equations*, BIT, 40 (2000), pp. 241–266.

[10] F. Filbet and L. Pareschi, *A numerical method for the accurate solution of the Fokker-Planck-Landau equation in the nonhomogeneous case*, Journal of Computational Physics, 179 (2002), pp. 1–26.

[11] M. Gander and E. Hairer, *Nonlinear convergence analysis for the parareal algorithm*, Lecture Notes in Computational Science and Engineering, 60 (2008), p. 45.

[12] M. Gander and S. Vandewalle, *Analysis of the parareal time-parallel time-integration method*, SIAM J. Sci. Comput., 29 (2007), pp. 556–578.

[13] ———, *On the superlinear and linear convergence of the parareal algorithm*, Lecture Notes in Computational Science and Engineering, 55 (2007), p. 291.

[14] G. Gilboa and S. Osher, *Nonlocal operators with applications to image processing*, CAM Report 07-23, UCLA, 2007.

[15] S. Gottlieb, D. Ketcheson, and C. Shu, *High order strong stability preserving time discretizations*, Journal of Scientific Computing, 38 (2009), pp. 251–289.

[16] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, Journal of Computational Physics, 73 (1987), pp. 325–348.

[17] T. Hagstrom and R. Zhou, *On the spectral deferred correction of splitting methods for initial value problems*, Commun. Appl. Math. Comput. Sci., 1 (2006), pp. 169–205.

[18] J. Huang, J. Jia, and M. Minion, *Accelerating the convergence of spectral deferred correction methods*, J. Comput. Phys., 214 (2006), pp. 633–656.

[19] ———, *Arbitrary order Krylov deferred correction methods for differential algebraic equations*, J. Comput. Phys., 221 (2007), pp. 739–760.

[20] D. I. Ketcheson, C. B. Macdonald, and S. Gottlieb, *Optimal implicit strong stability preserving Runge–Kutta methods*, Appl. Numer. Math., 59 (2009), pp. 373–392.

[21] A. Layton, *On the choice of correctors for semi-implicit Picard deferred correction methods*, Applied Numerical Mathematics, 58 (2008), pp. 845–858.

[22] A. Layton and M. Minion, *Implications of the choice of predictors for semi-implicit Picard integral deferred corrections methods*, Comm. Appl. Math. Comput. Sci., 1 (2007), pp. 1–34.

[23] A. T. Layton and M. L. Minion, *Implications of the choice of quadrature nodes for Picard integral deferred corrections methods for ordinary differential equations*, BIT, 45 (2005), pp. 341–373.

[24] J. Lions, Y. Maday, and G. Turinici, *A "parareal" in time discretization of PDEs*, Comptes Rendus de l'Academie des Sciences Series I Mathematics, 332 (2001), pp. 661–668.
[25] B. Lu, Y. Zhou, M. Holst, and J. McCammon, *Recent progress in numerical methods for the Poisson–Boltzmann equation in biophysical applications*, Comput. Phys, 3 (2008), pp. 973–1009.
[26] T. Macdougall and J. H. Verner, *Global error estimators for order 7, 8 Runge–Kutta pairs*, Numer. Algorithms, 31 (2002), pp. 215–231. Numerical methods for ordinary differential equations (Auckland, 2001).
[27] Y. Maday and G. Turinici, *A parareal in time procedure for the control of partial differential equations*, Comptes rendus-Mathématique, 335 (2002), pp. 387–392.
[28] M. Minion and S. Williams, *Parareal and spectral deferred corrections*, in NUMERICAL ANALYSIS AND APPLIED MATHEMATICS: International Conference on Numerical Analysis and Applied Mathematics 2008. AIP Conference Proceedings, vol. 1048, 2008, pp. 388–391.
[29] M. L. Minion, *Semi-implicit spectral deferred correction methods for ordinary differential equations*, Commun. Math. Sci., 1 (2003), pp. 471–500.
[30] W. Miranker and W. Liniger, *Parallel methods for the numerical integration of ordinary differential equations*, Mathematics of Computation, (1967), pp. 303–320.
[31] J. Nievergelt, *Parallel methods for integrating ordinary differential equations*, Communications of the ACM, 7 (1964), pp. 731–733.
[32] C. Runge, *Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten*, Zeit. für Math. und Phys., 46 (1901), pp. 224–243.
[33] G. van Rossum et al., *The Python programming language*. `http://www.python.org`, 1991. Accessed 2008-07-15.
[34] J. Verboncoeur, *Particle simulation of plasmas: review and advances*, Plasma Physics and Controlled Fusion, 47 (2005), p. 231.