Hybrid Dynamic Iterations for the Solution of Initial Value Problems

Yanan Yu and Ashok Srinivasan

Department of Computer Science, Florida State University, Tallahassee, FL32306

Abstract

Many scientific problems are posed as Ordinary Differential Equations (ODEs). A large subset of these are initial value problems, which are typically solved numerically. The solution starts by using a known state-space of the ODE system to determine the state at a subsequent point in time. This process is repeated several times. When the computational demand is high due to large state space, parallel computers can be used efficiently to reduce the time to solution. Conventional parallelization strategies distribute the state space of the problem amongst cores and distribute the task of computing for a single time step amongst the cores. They are not effective when the computational problems have fine granularity, for example, when the state space is relatively small and the computational effort arises largely from the long time span of the initial value problem. We propose a hybrid dynamic iterations method[1] which combines conventional sequential ODE solvers with dynamic iterations to parallelize the time domain. Empirical results demonstrate a factor of two to four improvement in performance of the hybrid dynamic iterations method over a conventional ODE solver on an 8 core processor. Compared to Picard iterations (also parallelized in the time domain), the proposed method shows better convergence and speedup results when high accuracy is required.

Keywords: time parallelization, hybrid dynamic iterations, ODE solver

Email address: `{yu,asriniva}@cs.fsu.edu` (Yanan Yu and Ashok Srinivasan)

[1]This paper is an extended version of a conference paper [1]. In this paper, we have considered an additional underlying ODE solver for the hybrid method, empirically evaluated with more ODE systems, and also evaluated the relative performance with low and high accuracy requirements.

1. Introduction

Many scientific computations involve solving initial value Ordinary Differential Equation (ODE) problems. The problems are solved by iteratively computing the state of the physical system described by the differential equations, until the required time span is covered. The computational effort of solving initial value problems arises from two aspects: 1) the state space of the system is large; 2) the time span of the problem is long. A common strategy to deal with the great computational effort is through some type of parallelization. Conventional approaches parallelize the computation by decomposing the state space of a system into smaller sub-domains, and they are effective for large scale systems.

A current technological trend has resulted in the need for alternative approaches to parallelization, as discussed below. In recent years, there has been a trend for providing increased computing power through greater number of cores on a chip, rather than through higher clock speeds. Desktops are equipped with multiple cores. Parallel processing is no longer restricted to the HPC community. Non-HPC users often have a need to solve smaller problems than the HPC community. For such applications, the speed of solution can still be important. For example, reducing the time to solve an ODE from a minute to around ten seconds is a noticeable difference in solution time. Any available parallelism should be efficiently exploited in order for better performance. However, for small systems in which the computational effort is caused predominantly by the large number of time steps (long time span), the conventional decomposition methods are less effective, as the communication overhead due to the synchronization at the end of each time step will dominate the computation cost of each sub-domain in the decomposition.

A possible solution to tackle the computational effort arising from the long time scale is to directly parallelize the time domain. The idea is to divide the entire time span of the simulation into smaller intervals and have each core simulate one interval at a time. However, time is an intrinsically sequential quantity. In proposing a time parallel method, the difficulty lies in that each time interval must complete before the next one can start, since we are solving an initial value problem. This leads to an essentially sequential process.

A few methods have been proposed by others to parallelize the time domain of initial value problems. Dynamic iterations [2, 3], were well studied in the 1980s and 1990s. These methods start out with a guess for the solution over the entire time span, and in each iteration, they refine the approximate solution from the previous iteration. They are parallelized by dividing the time span into time intervals, and having each core update a different time interval in each iteration. These methods have a couple of limitations. The first is that they have high memory requirements. The second is that their convergence is slow for many realistic ODEs, which leads to low parallel efficiency. With multicore processors widely available, we believe it is time to reevaluate the applicability of dynamic iterations to solving initial value problems, especially in situations where conventional methods do not yield any speedup. We show in our empirical results that Picard iterations, a type of dynamic iterations, achieve a factor of two improvement in speed over a conventional ODE solver on an 8-core processor when low accuracy is acceptable.

In this paper, we propose an improved dynamic iterations method, namely hybrid dynamic iterations. The hybrid dynamic iterations method combines the conventional sequential ODE solver with dynamic iterations. The method guarantees that at least one time interval converges in each iteration, and thus the total computation time of the hybrid method is no worse than an equivalent ODE solver, apart from a small parallelization overhead. Empirical results demonstrates a factor of two to four improvement in performance of the hybrid dynamic iterations method over an equivalent conventional sequential solver on an 8-core processor. Compared to Picard iterations (also parallelized in the time domain), the proposed method shows better convergence and speedup results when high accuracy is required.

2. Hybrid Dynamic Iterations

2.1. Dynamic Iterations

In this section, we consider the problem of solving an Initial Value Problem involving first order Ordinary Differential Equations (ODE). A simple description of the problem and the conventional approach to solving it are given below. A first order ODE is often given in the form,

$$\dot{u} = f(t, u), u(0) = u_0 \tag{1}$$

where u is the state of a physical system and may be a vector, and t is typically time. The initial state of the system, $u(t_0)$ at time t_0, is provided in an Initial Value Problem. The problem is to determine the value of u at subsequent values of time, up to some time span T. An ODE solver iteratively determines the values of u by starting from the known initial value $u(t_0)$, and uses $u(t_i)$ to find $u(t_{i+1})$. In this paper, we consider the first order ODEs. Higher order ODEs can be expressed as a first order ODE system using a standard transformation [4].

For small ODE systems, the solution time may still be significant if the number of time steps is large, for example, the problem is solved for a long time span. Direct parallelization of the time domain appears promising then. Dynamic iterations are a class of methods that are often suitable for time parallelization. The idea of dynamic iterations is to start out with an initial guess for the entire time domain, and then iteratively improve the solution until it converges. Dynamic Iterations solve eqn. 1 recursively with the following equation,

$$\dot{u}^{m+1} - g(u^{m+1}, u^m) = f(t, u^m), u^{m+1}(0) = u_0 \tag{2}$$

The recursion starts with the initial guess $u^0(t) = u_0$ and g is defined such that $g(u, u) = 0$. If the iterations converge, they converge to the exact solution of f, provided that eqn. 2 is solved exactly [3]. The choice of g affects the rate of convergence. In practice, numerical methods are always used to solve eqn. 2, and g also affects the cost of the numerical integration.

Picard iterations, which predate modern ODE solvers by a few decades, is a special case of dynamic iterations. If g in eqn. 2 is chosen such that $g(y, z) = 0$, then the solver is equivalent to the Picard iterations. We have $\dot{u}^{m+1} = f(t, u^m)$. The solution is found by iteratively solving a sequence of equivalent integrals. Initially, the value of u^0 is stored at discrete time points for the entire time domain. In the $(m+1)$th iteration, the first step is to evaluate $\dot{u}^{m+1}$ by $f(t, u^m)$. Then f is numerically integrated using some quadrature method. This leads to the following expression for determining the value of u^{m+1}, when the integration is exact.

$$u(t)^{m+1} = u_0 + \int_0^t f(s, u^m(s))\, ds. \tag{3}$$

Picard iterations can be easily parallelized in time. Let t_i be a sequence of time intervals and u_i^0 denote the value of u at the ith interval, where $i \in (0..n)$ and n is the total number of the time intervals. In each of the interval of (t_i, t^{i+1}), the values of u is known for several discrete time points.

We can rewrite the above equation as follows.

$$u_{i+1}^{m+1} = u_i^{m+1} + \int_{t_i}^{t_{i+1}} f(s, u^m(s))\, ds \tag{4}$$

Core i, $0 \leq i \leq P$ is responsible for determining the values of u^{m+1} in the interval $(t_i, t_{i+1}]$. First, it evaluates the integrals in eqn. 4 for the discrete time points using some quadrature method, for which it also needs some boundary values of u^m from neighbor cores. Then a parallel prefix computation collects the integrals from each core and computes the cumulative integrals from the interval t_0 to t_{n-1}. After this, each core independently updates the value of u^{m+1} for the discrete time points in its interval. In this parallel scheme, communication is required by the parallel prefix for updating the boundary values of each time interval. Therefore, parallelization is efficient only if each time interval contains sufficient time points for evaluating the integral in eqn. 4 and the computation cost is much larger than the prefix overhead.

However, they were not suitable for most of the realistic problems due to their slow convergence. Picard iterations also impose a high memory requirement, because the solution at all time steps in the previous iteration is needed for the update in the next iteration. For example, in order to simulate n time steps, all the values of u^m at the n time points need to be stored. Usually the total time domain is divided into smaller windows, and the solution for one window converges before computing for the next window. However, the number of time steps in each window can still be high. The performance will also be affected by the increased number of cache misses.

2.2. A Hybrid Scheme

This paper proposes a hybrid scheme, which combines exact ODE solvers with dynamic iterations. The hybrid method is presented as an intuitive improvement to Picard iterations to deal with one of the latter's shortcoming – its high memory requirement.

Consider the initial value problem defined in eqn. 1. The exact solution is given as follows,

$$u_{i+1} = u_i + \int_{t_i}^{t_{i+1}} f(s, u(s))ds \tag{5}$$

If we compare the above exact solver to the Picard iterations, as in eqn. 4, we see that Picard iterations replace u with u^m in the integral, where u^m is the latest approximation to u in the iterative process. The values of u^m at each time step in $[t_i, t_{i+1}]$ need to be stored for the next iteration.

In the hybrid method, each core solves the exact ODE as in eqn. 5 with initial condition $\hat{u}_i(t_i) = u_i^m$ at time t_i, up to time t_{i+1} as in eqn. 6, where $i \in [1 \dots P]$ and P is the total number of cores.

$$\hat{u}_i(t_{i+1}) = \hat{u}_i(t_i) + \int_{t_i}^{t_{i+1}} f(s, \hat{u}_i(s))ds \tag{6}$$

The hat on u indicates that the solution with an exact ODE solver might start from a possibly incorrect initial condition. We then have the following expression for the $(m+1)$th iteration,

$$\hat{u}_i(t_{i+1}) - u_i^m = \int_{t_i}^{t_{i+1}} f(s, \hat{u}_i(s))ds \tag{7}$$

where, the initial condition is assumed as $\hat{u}_i(t_i) = u_i^m$ after the mth iteration at time t_i. If we replace the integral in the Picard iterations recurrence in eqn. 4 with the integral in eqn. 7, and in turn, replace the integral with the integral with the left hand side of the expression in eqn. 7, we get the recurrence for the hybrid method as follows.

$$u_{i+1}^{m+1} = u_i^{m+1} + \hat{u}_i(t_{i+1}) - u_i^m \tag{8}$$

Fig. 1 shows a schematic of the hybrid dynamic iterations method. Each core starts from some initial state u_i^m, represented by the dashed line and computes $\hat{u}_i(t_{i+1})$ by solving the ODE system independently. The results are shown by the solid line with arrow. A parallel prefix is then performed to compute the new approximation of u_{i+1}^{m+1}, represented by the dash-dot-dot line. The process repeats until u_i converges. An intuitive way of thinking about the hybrid method is as follows. If u_i^m is accurate, then $\hat{u}_i(t_{i+1})$ is the exact solution for u at time t_{i+1}, and u_{i+1}^{m+1} converges to u_{i+1}^m. However, most likely the initial states are incorrect, since they can only be approximated without solving the ODE system exactly for the entire time domain. We denote the differences observed for u at time t_i between two successive iterations as $e_i = u_i^{m+1} - u_i^m$. We add e_i back to $\hat{u}_i(t_{i+1})$ as a corrector and get u_{i+1}^{m+1}. Algorithm 2.1 gives a formal description of the parallel hybrid method.

If the iterations in the hybrid method converge, then they converge to the exact solution of the ODE system, provided that the ODE is solved

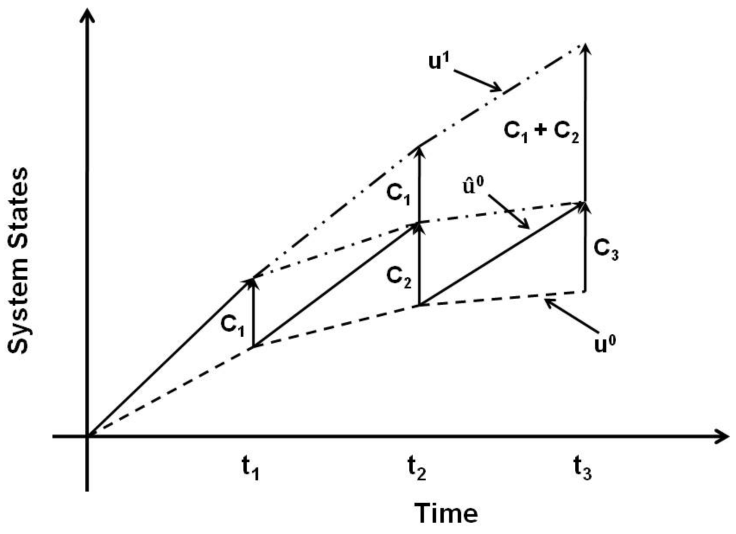


Figure 1: A schematic of hybrid dynamic iterations.

exactly in each iteration. Of course, the ODE is actually solved numerically, with approximation errors. At convergence, $u_i^{m+1} = u_i^m$, for $i \in [1, P]$. From eqn. 8, we then have $u_{i+1}^{m+1} = \hat{u}_i(t_{i+1})$. Since core 0 starts from the exact initial condition which is given, $u_1^{m+1} = \hat{u}_0(t_1)$ is always the exact solution. Because u converges and $u_1^{m+1} = u_1^m$, u_1^m is also exact. Since the computation on core 1 is started from the exact u_1^m to determine the value $\hat{u}_1(t_2)$, $u_2^{m+1} = \hat{u}_1(t_2)$ is exact too. By induction, we can show that u_i^{m+1} on all the cores are the exact solution to the ODE.

Algorithm 2.1: HYBRID-METHOD($u_i^0 = u_0$, Number of cores P)

```
repeat
  do
    for each core $i \in \{0..P-1\}$
      do
        $\hat{u}_{(i)}^m(t_{i+1}) \leftarrow$ Solve ODE with initial condition $u_i^m$)
        $\Delta_i \leftarrow \hat{u}_{(i)}^m(t_{i+1}) - u_i^m$
    $G_i \leftarrow$ Parallel prefix on $\Delta_i$
    $u_{i+1}^{m+1} \leftarrow G_i + u_0$
until Convergence
```

If the ODE solver is exact, it can be proved that the hybrid method converges in a finite number of iterations. In each iteration, note that core 0 always starts from the exact initial condition u_0 and thus the solution on core 0 converges after the first iteration. In the second iteration, the initial condition on core 1 is also exact (as u_1^1 from core 0), and so is the solution u_2^2 for time t_2 on core 1. By induction, the hybrid method will converge in n iterations if there are n intervals to solve for the entire time domain of the ODE system, which is the worst case for the parallel hybrid method. In other words, the solution converges at most n iterations, and the parallel performance will not be worse than an equivalent conventional sequential solver, if the parallel overhead is small. Of course, if the parallel hybrid method solves the ODE in n iterations, no speed-up would be gained over an equivalent sequential ODE solver. In practice, we expect that the hybrid method converges in less than n iterations.

Compared to Picard iterations, the hybrid method is expected to converge faster. The hybrid method differs from Picard iterations in how the term $\int_{t_i}^{t_{i+1}} f(s, u(s))ds$ in eqn. 5 is estimated, without knowing the actual solution of u. Picard iterations perform a quadrature computation with the approximation of u^m to get u^{m+1}. In the hybrid method, though the initial condition u^m may also be incorrect, the integration is computed with an exact ODE solver. Intuitively speaking, we expect that the solution in the hybrid method be better than Picard iterations, which always use the old approximation of u^m for the entire time domain from the previous iteration to compute u^{m+1}. Also note that the memory consumption of the hybrid method is small - only the results at certain time points t_i are stored. We do not need to store the results at all intermediate time steps in the time interval of $[t_i, t_{i+1}]$ in eqn. 6, whereas the values at all the intermediate steps are necessary for Picard iterations to numerically evaluate the quadrature. Unlike Picard iterations, the memory requirement by the hybrid method is roughly the same as a sequential solver and it is independent of the size of the time step and the number of time steps in a time interval.

3. Empirical Evaluation

3.1. Numerical Examples

We consider six first order ODE examples. The first three are linear, and the rests are nonlinear. The second, fourth and sixth systems are also used in [5, 6] as the test problems for comparing different numerical methods. The exact solutions to the six ODEs are shown in fig. 2.

Example ODEs:

1. The initial value problem:

$$\begin{cases} y_1\prime = y_2 & y_1(0) = 0 \\ y_2\prime = -y_1 & y_2(0) = 1 \end{cases}$$

   The exact solution for this system is, $y_1 = \sin(t)$, $y_2 = \cos(t)$.

2. The initial value problem:

$$\begin{cases} y_1\prime = y_1 + y_2 & y_1(0) = 0 \\ y_2\prime = -y_1 + y_2 & y_2(0) = 1 \end{cases}$$

   The exact solution for this system is, $y_1 = \sin(t)e^t$, $y_2 = \cos(t)e^t$

3. The initial value problem:

$$\begin{cases} y_1\prime = y_2 & y_1(-6) = 1 \\ y_2\prime = -y_1 t & y_2(-6) = 1 \end{cases}$$

   It is also known as the Airy equation, which is seen in physics and astronomy.

4. The initial value problem:

$$\begin{cases} y_1\prime = 2y_2^2 & y_1(0) = 1 \\ y_2\prime = e^{-t}y_1 & y_2(0) = 1 \\ y_3\prime = y_2 + y_3 & y_3(0) = 0 \end{cases}$$

   The exact solution for this system is, $y_1 = e^{2t}$, $y_2 = e^t$, $y_3 = te^t$.

5. The initial value problem:

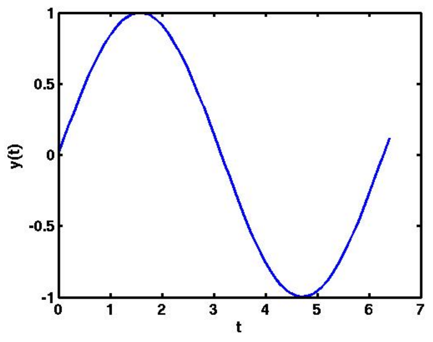
$$\begin{cases} y_1\prime = y_2 & y_1(-6) = 1 \\ y_2\prime = \gamma\cos(\omega + \phi) - \delta y_2 - \beta y_1^3 - \omega_0^2 y_1 & y_2(-6) = 1 \end{cases}$$

   where, $\gamma = 2.3$, $\omega = 1$, $\phi = 0$, $\delta = 0.1$, $\beta = 0.25$, $\omega_0 = 1$.
   This ODE system is known as the Duffing equation and it is often used to describe the motion of a damped oscillator.

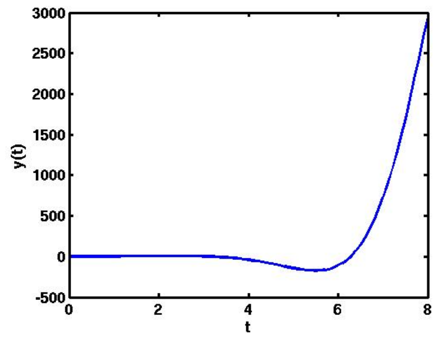
6. The initial value problem:

$$\begin{cases} y_1\prime = -y_1 & y_1(0) = 1 \\ y_2\prime = y_1 - y_2^2 & y_2(0) = 0 \\ y_3\prime = y_2^2 & y_3(0) = 0 \end{cases}$$

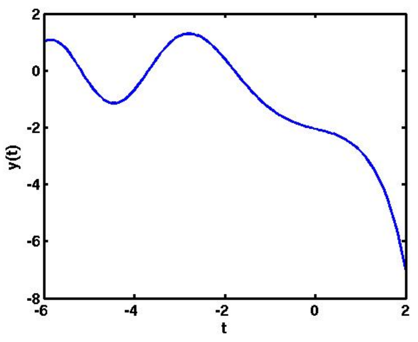
   This nonlinear system represents a nonlinear chemical reaction [6].



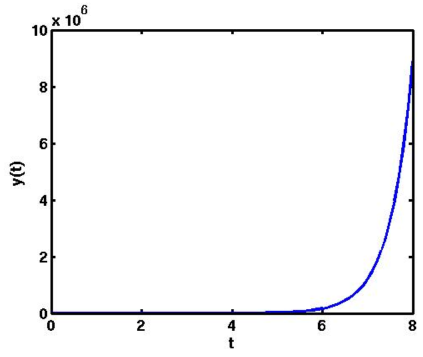
(a) ODE1



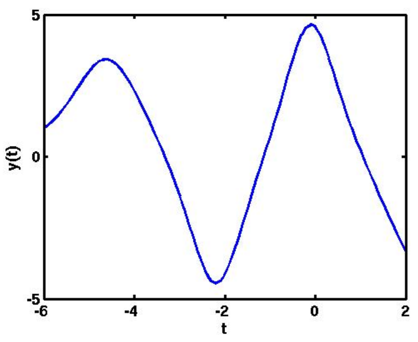
(b) ODE2



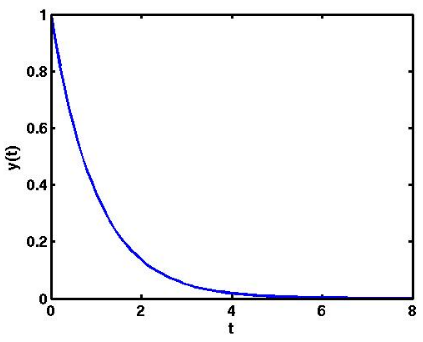
(c) ODE3



(d) ODE4



(e) ODE5



(f) ODE6

Figure 2: Exact solutions of the six example ODEs.

In the experimental results, we also need to report the empirical global errors of the hybrid method. Since there are no closed form solutions to the Airy function, the Duffing function and ODE6, the exact solutions are approximated by the results from *ode45* in Matlab, with both the absolute error and relative error set to 10^{-12}. In the experiments, we choose the step size of an ODE solver such that the global error is of the order of 10^{-6}; so, the accuracy in the approximated exact solution is sufficient for the comparisons in this report.

3.2. Experiment Setup

The experimental tests are performed on a two-socket Quad-core Intel Xeon processor with 8GB memory. OpenMPI 1.2 is used for inter-process communications. Both the sequential code and parallel code are compiled using *gcc* with highest optimization level of the compiler. The speedup is reported by comparing the parallel result with that of equivalent conventional sequential solvers; parallelization of the conventional solvers, with either OpenMP or MPI, led to a slow-down in performance. An assembly timing routine, which accesses the time-stamp counter through the *rdtsc* instruction on Intel processors is used for timing the results. The resolution of the timer is 0.05μs, which is sufficiently small for the timing purpose in this paper, since the timing results are at least the order of a few microseconds.

In order to compare the hybrid method and Picard iterations, the ODE solver used by the hybrid method and the quadrature evaluation method in Picard iterations should have the same order of accuracy. A single-step 4th order Runge-Kutta method and a multi-step 4th order Adams-Bashforth-Moulton method are implemented as the underlying exact ODE solvers of the hybrid method. The Simpson's rule is used to evaluate the quadrature in Picard iterations. The accuracy of the Simpson's rule is also of order 4. All the computations are in double precision.

We also need to clarify the terms *tolerance* and *error* used in the discussion. The latter refers to the absolute difference between the exact solution and the computed solution. The term *tolerance* is a parameter passed to the iterative solver to determine convergence. When an iterative method is used, the solution is considered to have converged if the absolute difference between the two solutions at successive iterations is smaller than a threshold, which is referred to as the *tolerance*. The *error* may be larger than the *tolerance*. For a sequential ODE solver, *tolerance* has its conventional meaning.

3.3. Experiment Results

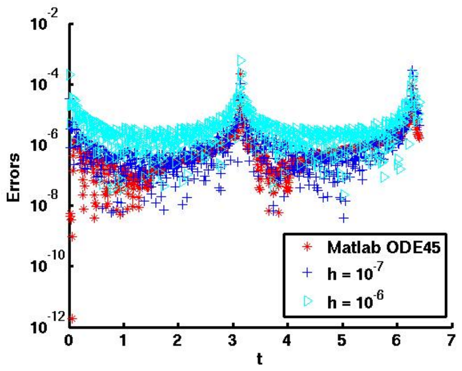
3.3.1. Choice of Step Size

We first explain our choices of the time step size for the exact ODE solvers in the hybrid method. The step size has to be small enough for stability and accuracy concerns. However, if the time step size is smaller than necessary, then the granularity of the computation is coarser, and thus the communication overheads will be artificially made small, relative to the computational effort in the parallel implementation. In order to avoid these artificial effects caused by too small time step size, we first determine the accuracies of the sequential ODE solver and quadrature evaluation for different step sizes, and we choose the one that gives sufficiently good accuracy. We also compare the exact solutions with the results from using Matlab $ode45$, a medium order ODE solver, with both the relative error tolerance and absolute error tolerance set equal to 10^{-6}. We choose the step size such that the errors are the same order as that of Matlab $ode45$, approximately the order of 10^{-6}.

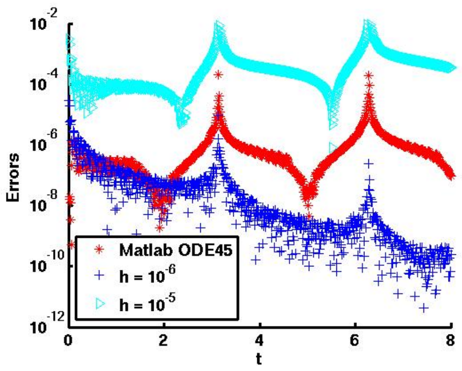
For Runge-Kutta 4th order method, for example, a step size of 0.005 is sufficiently small for solving ODE1 stably. However, a much smaller step size is usually required for high accuracy. Fig. 3 (a)-(f) compare the errors in the solutions from $ode45$ function in Matlab (with parameters of absolute error $= 10^{-6}$ and relative error $= 10^{-6}$) and Runge-Kutta 4th order method with two different time step sizes for the six ODE examples. For ODEs (1, 3, 5, 6), the step size is determined as 10^{-7}, and as 10^-6 for ODEs (2,4). The same analysis and step sizes are also used in the multi-step Adams-Bashforth-Moulton method and the Simpson's rule for the quadrature evaluations in Picard iterations, as they are all the same order with respect to time step size.

3.3.2. Effects of Windowing

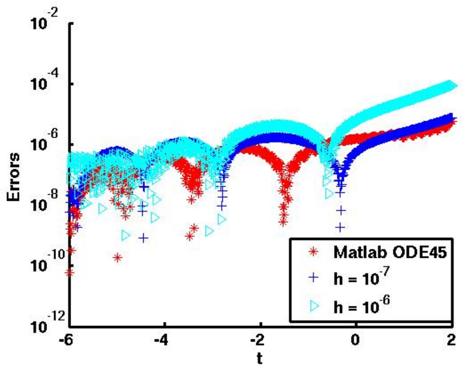
Three variants of the hybrid method are considered: (i) The entire time span T of the ODE system is divided into T/P intervals on P cores and each core solves for one time interval; (ii) T is divided into a number of windows of width W, and the number of windows is much larger than P. The cores solve the ODE system for the first P windows (one window per core) and proceed to the next P windows only if all the cores have converged. This type of windowing technique has been shown to be more effective than solving for the entire time domain in dynamic iterations [7, 8]. (iii) In the above windowing method, after the first few cores converge, they no longer perform any useful work rather than repeating the computation. These cores,



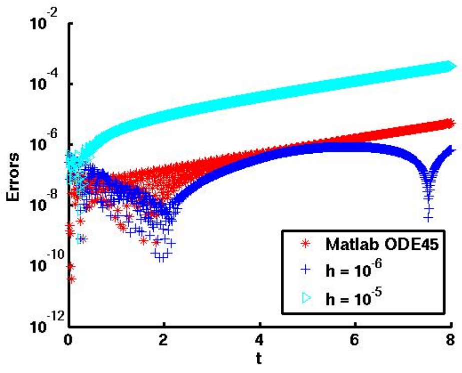
(a) ODE1



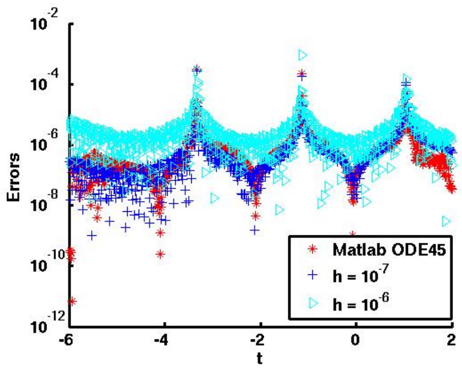
(b) ODE2



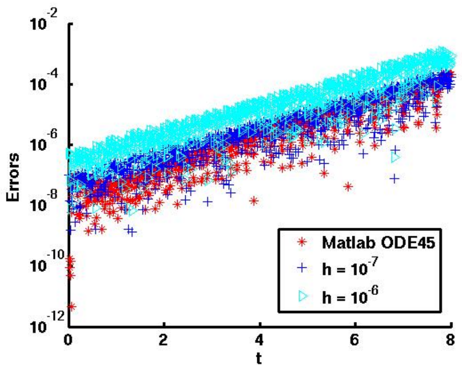
(c) ODE3



(d) ODE4

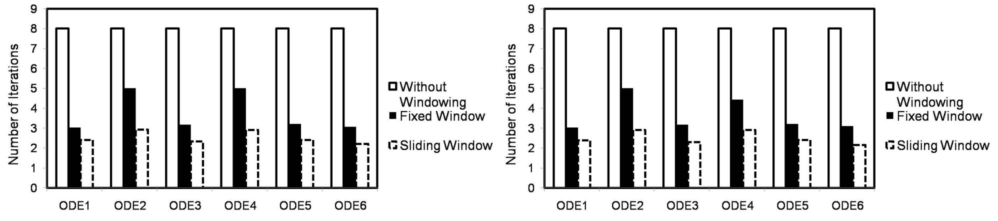


(e) ODE5



(f) ODE6

Figure 3: Accuracy of 4th order Runge-Kutta method with different step sizes.



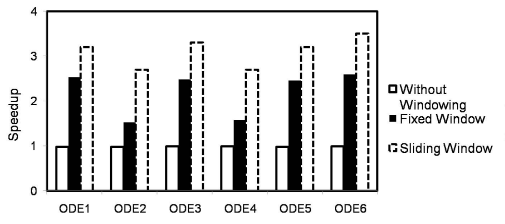
(a) Runge-Kutta Method (b) Adams-Bashforth-Moulton Method

Figure 4: Number of iterations per window.

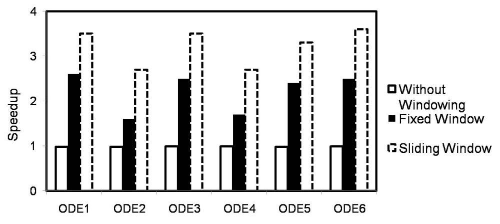
instead, can be used for computing new windows. We call this method *sliding windows*. We also note that a core cannot be considered as having converged, unless it has locally converged and all the cores handling smaller values of the time intervals have also converged.

Fig. 4 shows the number of iterations per window from using the three different windowing methods on 8 cores with Runge-Kutta and Adams-Bashforth-Moulton methods. Fig. 5 compares the speedup results for the six example ODEs. For non-windowing method, each core computes for about 10^7 time steps. For the two windowing methods, the window size is 10^4 time steps and the tolerance is 10^{-6}. As we can see, sliding windows yield a better speedup than fixed windows, since once a core has converged, it will be used for new computation. The non-windowing method does not perform well for such small number of cores, because eight iterations are required to converge for the given time span on eight cores. Given these observations, we will only use the results from the sliding windows method for the discussions that will follow.

Speedup Results. The speedup results are reported as comparing the hybrid method and Picard iterations with a conventional ODE solver. If we compare against a sequential implementation of dynamic iterations, speedup is close to linear. However, in order to realistically evaluate the usefulness of dynamic iterations, we need to compare them against a conventional ODE solver. The total computational work of a sequential solver is less than that of dynamic iterations, and so the speedup results are lower than if compared with a sequential implementation of dynamic iterations.



(a) Runge-Kutta Method

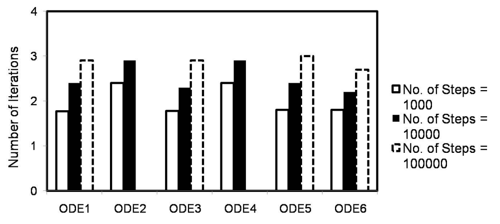


(b) Adams-Bashforth-Moulton Method

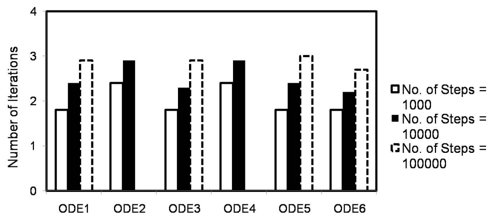
Figure 5: Speedup results.

The hybrid method is implemented with MPI for inter-process communications. In particular, the method with sliding windows requires three MPI function calls per iteration: 1) MPI_Scan is used for the parallel prefix summation; 2) MPI_Allreduce is called in testing for convergence; 3) MPI_Isend and MPI_Recv communicate the latest updated value between certain cores. Across eight cores, MPI_Scan, MPI_Isend and MPI_recv take less than 1μs and MPI_Allreduce takes less than 10μs. The computation time per step for the example ODEs using Runge-Kutta method or Adams-Bashforth-Moulton method takes less than 1μs. For speedup results shown in the following discussions, the window size is at least 10^3 time steps, in which case, the granularity of the computation is in the order of a hundred microseconds, and the overall communication overhead is at most the order of 10%.

We now discuss the choice of window size. As mentioned earlier, the window size used in the hybrid method directly determines the granularity of the parallel computation. Smaller granularity indicates relatively larger communication overhead, and the speedup can be expected to decrease as the window size becomes smaller. In the hybrid method, the window size can also affect the total number of iterations for convergence. Larger number of iterations will lead to lower speedup. We experimented with three different window sizes and each includes 10^3, 10^4 and 10^5 time steps respectively. Fig. 6 compares the average number of iterations per window with two underlying conventional ODE solvers. The tolerance for convergence is 10^{-6}. Fig. 7 shows the speedup when different window sizes are used for the example ODEs with same tolerance. The empirical results show that the smaller the window size is, the faster the method converges.

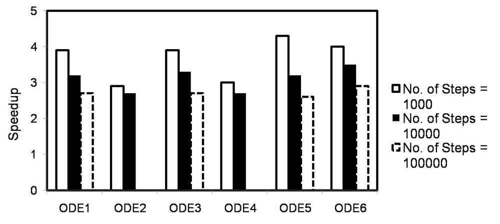


(a) Runge-Kutta Method

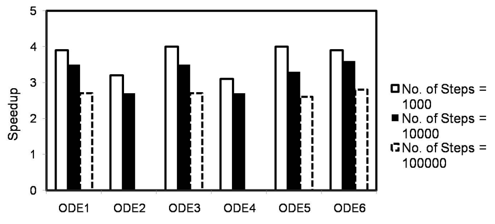


(b) Adams-Bashforth-Moulton Method

Figure 6: Number of iterations per window. Error tolerance is 10^{-6}.

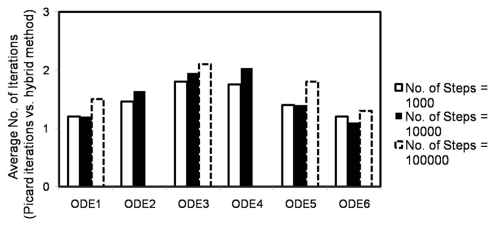


(a) Runge-Kutta Method

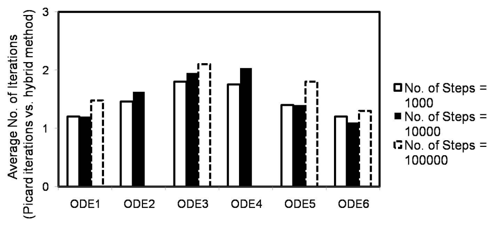


(b) Adams-Bashforth-Moulton Method

Figure 7: Speedup results for different window sizes. Error tolerance is 10^{-6}.



(a) Runge-Kutta Method



(b) Adams-Bashforth-Moulton Method

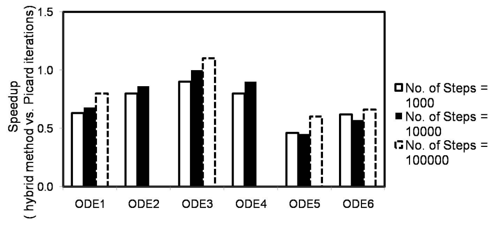
Figure 8: Comparison of the relative average number of iterations per window. The bars show the ratio obtained by dividing the results of Picard iterations against that of the hybrid method. Error tolerance is 10^{-6}.

Table 1: Ratio of the differences in computation time per time step. The computation time of Picard iterations is set as 1.

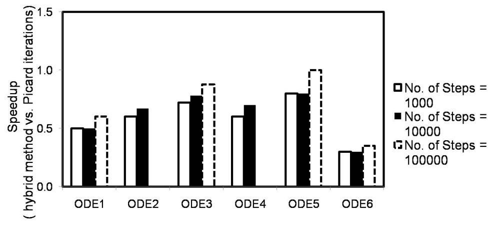
ODE	RK4	ABM	Simpson
ODE1	1.9	2.4	1
ODE2	1.9	2.4	1
ODE3	1.9	2.5	1
ODE4	2.2	2.9	1
ODE5	3.0	1.7	1
ODE6	1.9	3.7	1

Comparison with Picard iterations. Fig. 8 shows the comparisons of the number of iterations between the Picard iterations and the hybrid method for the same parameters of window size, time step size and error tolerance. Picard iterations in this paper are also implemented with sliding windows method. The step sizes used in Simpson's method to compute the quadrature integration of Picard iterations are the same as that used by the conventional ODE solvers in the hybrid method for each example ODE system. The error tolerance is also 10^{-6}. The y-axis shows the ratio of the number of iterations required by Picard iterations versus the number of iterations required by the hybrid method.

For the six example ODEs, Picard iterations require 1 - 2 times more iterations to converge than the hybrid method. Fig. 9 shows the comparison



(a) Runge-Kutta Method



(b) Adams-Bashforth-Moulton Method

Figure 9: Comparison of relative speedup results. The bars show the ratio of the speedup results of the hybrid method divided by that of the Picard iterations. Error tolerance is 10^{-6}.

in the speedup results between the hybrid method and Picard iterations on 8 cores. Note that the improvement in the convergence rate of the hybrid method does not directly translate to a proportional improvement in the speedup over Picard iterations. The disparity in speedup and convergence rate is due to the difference in the computation complexity of the exact ODE solvers (Runge-Kutta method and Adams-Bashforth-Moulton method) in the hybrid method and the integrator (Simpson's rule) in Picard iterations. For each time step, Runge-Kutta method and Adams-Bashforth-Moulton method require more computation time than the Simpson's rule. Table. 1 compares the differences in computation time per time step for the three methods and the differences are reported as the ratio of the differences compared to Picard iterations. Though Picard iterations need more iteration to converge, they achieve better speedup results. However, we also need to check the global error when the solution converges. The actual speedup of the two methods should be compared given that the solution has the same order of global errors as an equivalent sequential solver.

3.3.3. Accuracy of solution

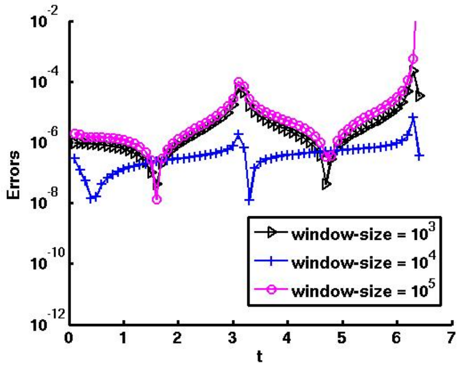
For dynamic iterations, both the window size and the error tolerance may affect the numerical accuracy of the final solution. Fig. 10 and fig. 11 show the global errors in the solution of the hybrid method with Runge-Kutta method. Similar results are also observed for the hybrid method with Adams-Bashforth-Moulton method. For different window sizes and different error tolerance, the accuracy of the solution varies. For example, if the window size is identical, then higher tolerance results in lower accuracy of the final solution.

The window size and the error tolerance also affect the accuracy and convergence rate of Picard iterations. Fig. 12 and fig. 13 show the global errors in the solution of Picard iterations for different window sizes and different values of tolerance. Compared to the hybrid method, for the same window size and same tolerance, Picard iterations have higher global errors. To obtain the global errors that are the same order as the hybrid method, Picard iterations would require lower error tolerance for convergence and thus more iterations.

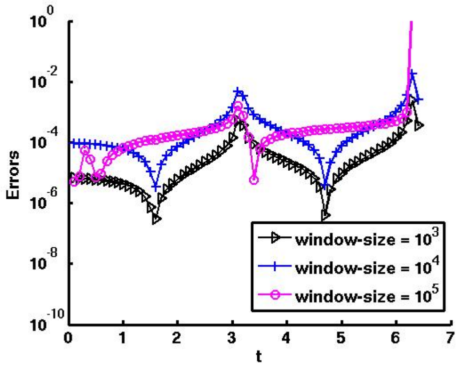
In order to compare the actually speedup of the hybrid method and Picard iterations over an equivalent sequential ODE solver, we need to compare the results when the hybrid method and Picard iterations have the same order of global error as the sequential ODE solver. We choose the window size and tolerance for both the hybrid method and Picard iterations such that the global error is in the order of 10^{-6}, as shown in fig. 10 - fig. 13. Fig. 14 shows the comparison of the speedup results between the hybrid method and Picard iterations, given the same order of global errors by the two methods. As we can see, the hybrid method with Runge-Kutta method is faster than Picard iterations, except for ODE5. The hybrid method with Adams-Bashforth-Moulton multistep ODE solver is faster than Picard iterations for half of the example ODEs.

3.3.4. Lower order of accuracy

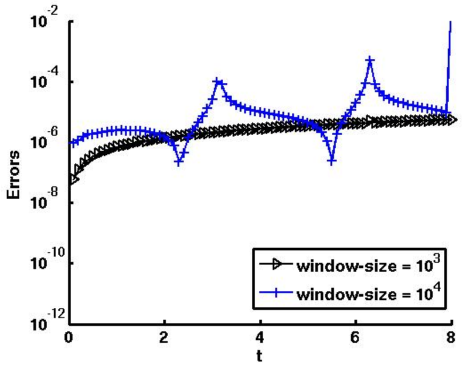
When higher order of global error is acceptable for solving an ODE system, Picard iterations can achieve better performance than a sequential ODE solver and the hybrid method on multicore processors. For example, if the sequential ODE solver uses time step size of 10^{-4} for the same six example ODEs, the global error is in the order of 10^{-3}. Since the window size in the hybrid method has to be at least in the order of 10^3 due to the parallel overheads and thus the total number of windows significantly decreases in the time span of the six ODE systems, the performance of the hybrid method is close to its equivalent sequential ODE solver. The same is also true for Picard iterations and Picard iterations actually require more than 8 iterations per window to converge on 8 cores. Though the total computation is larger in Picard iterations than a sequential ODE solver, Picard iterations can still expect some speedup because of the disparity in the computation time



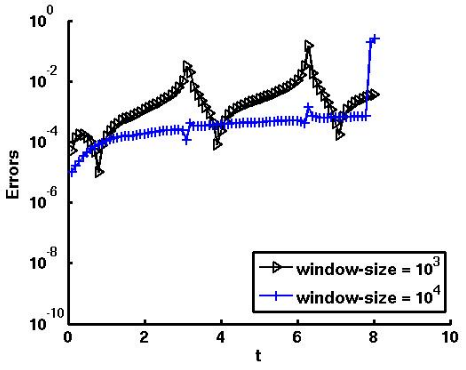
(a) ODE1. Tolerance = 10^{-6}



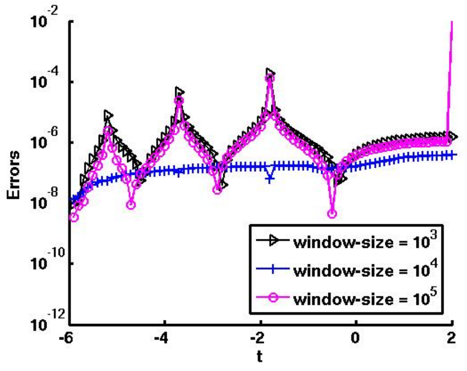
(b) ODE1. Tolerance = 10^{-4}



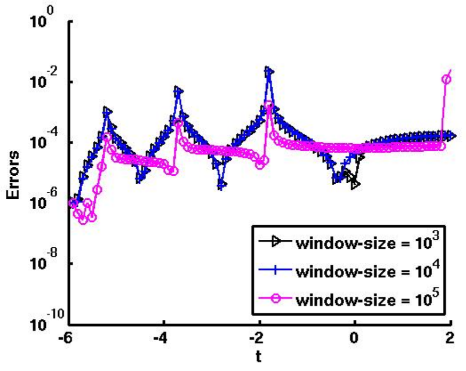
(c) ODE2. Tolerance = 10^{-6}



(d) ODE2. Tolerance = 10^{-4}

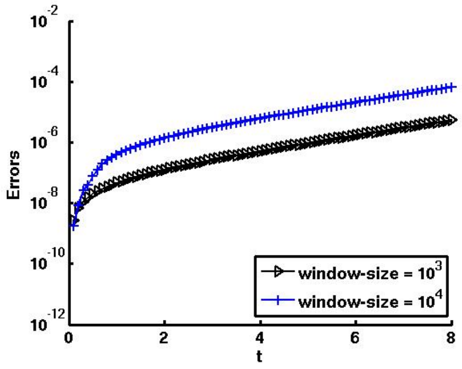


(e) ODE3. Tolerance = 10^{-6}

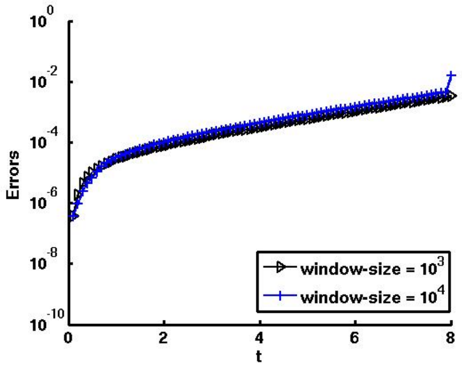


(f) ODE3. Tolerance = 10^{-4}

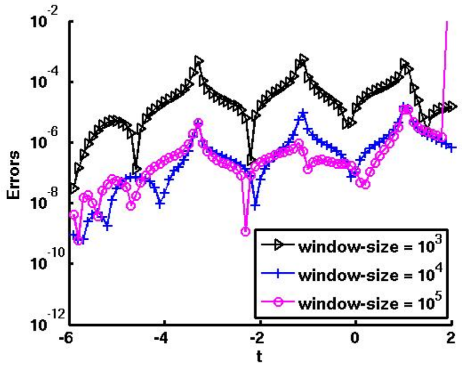
Figure 10: Global error in the solution of the hybrid method.



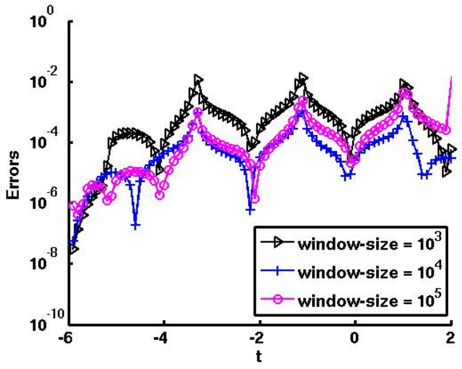
(a) ODE4. Tolerance = 10^{-6}



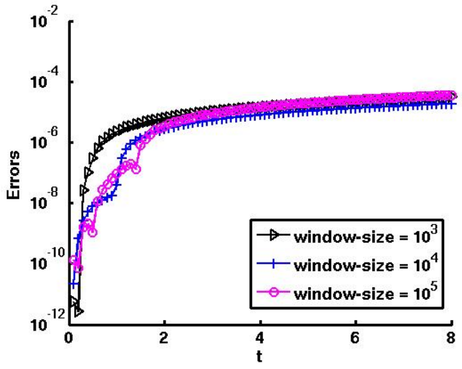
(b) ODE4. Tolerance = 10^{-4}



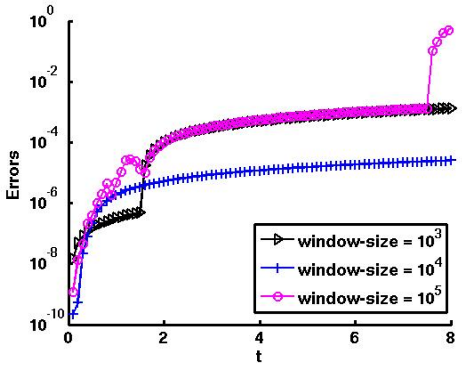
(c) ODE5. Tolerance = 10^{-6}



(d) ODE5. Tolerance = 10^{-4}

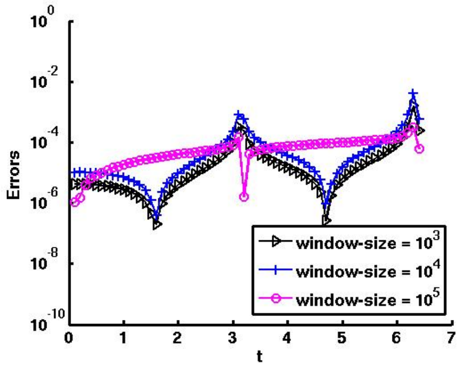


(e) ODE6. Tolerance = 10^{-6}

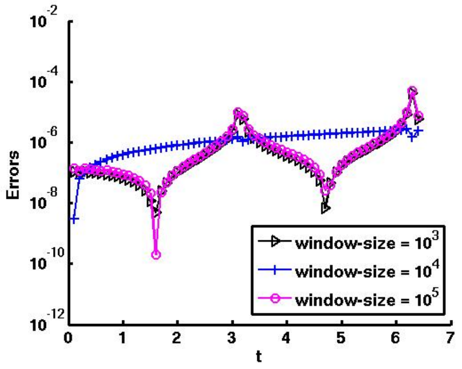


(f) ODE6. Tolerance = 10^{-4}

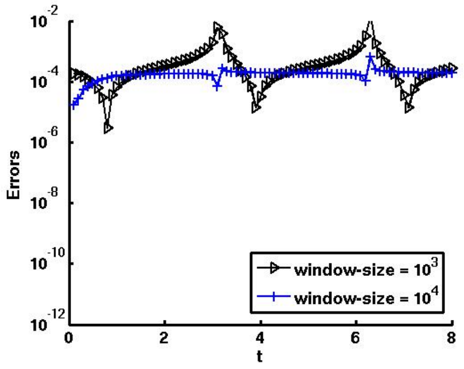
Figure 11: Global error in the solution of the hybrid method.



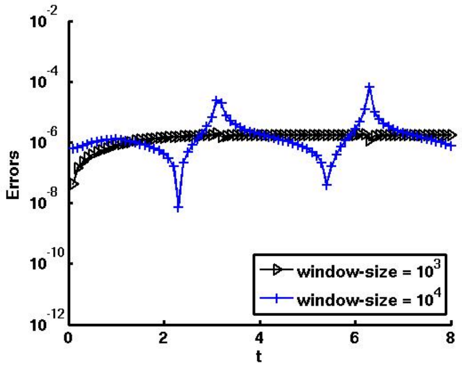
(a) ODE1. Tolerance = 10^{-6}



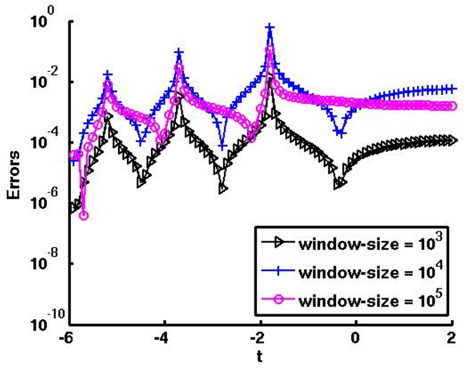
(b) ODE1. Tolerance = 10^{-7}



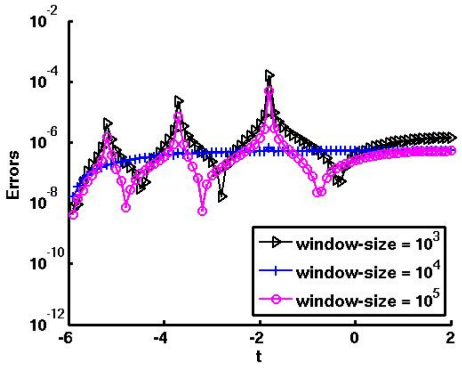
(c) ODE2. Tolerance = 10^{-6}



(d) ODE2. Tolerance = 10^{-8}

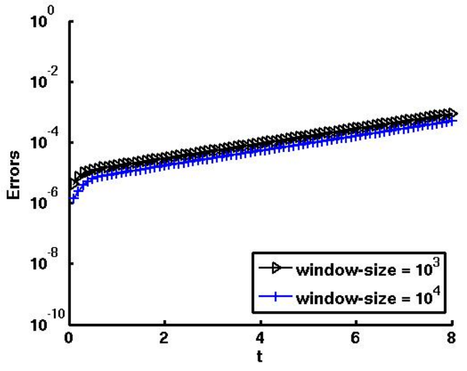


(e) ODE3. Tolerance = 10^{-6}

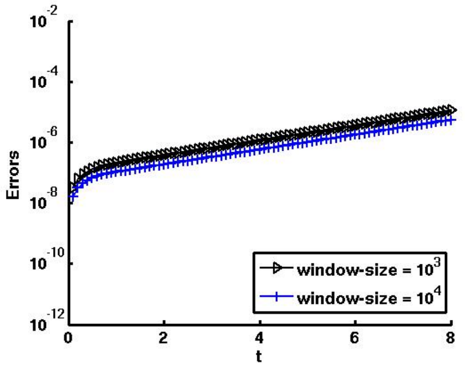


(f) ODE3. Tolerance = 10^{-8}

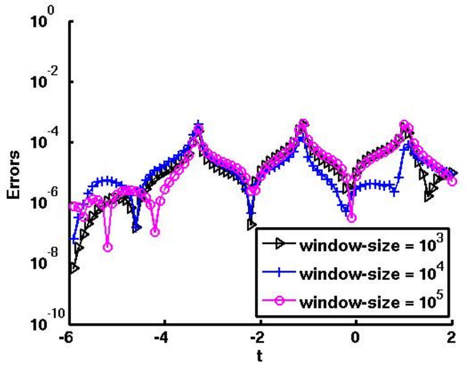
Figure 12: Global error in the solution of Picard iterations.



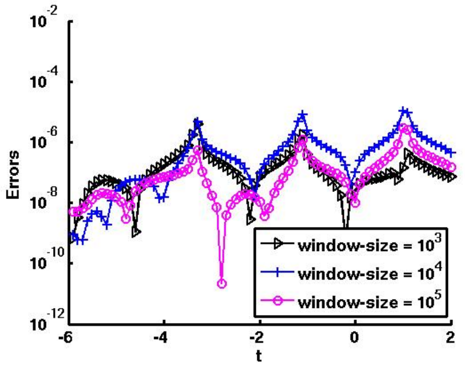
(a) ODE4. Tolerance = 10^{-6}



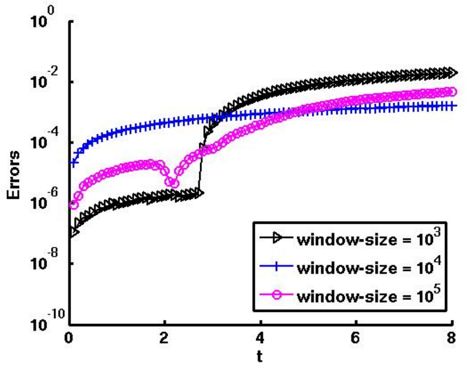
(b) ODE4. Tolerance = 10^{-8}



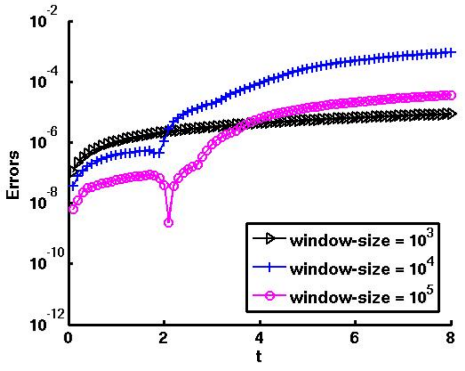
(c) ODE5. Tolerance = 10^{-6}



(d) ODE5. Tolerance = 10^{-8}



(e) ODE6. Tolerance = 10^{-6}



(f) ODE6. Tolerance = 10^{-8}

Figure 13: Global error in the solution of Picard iterations.

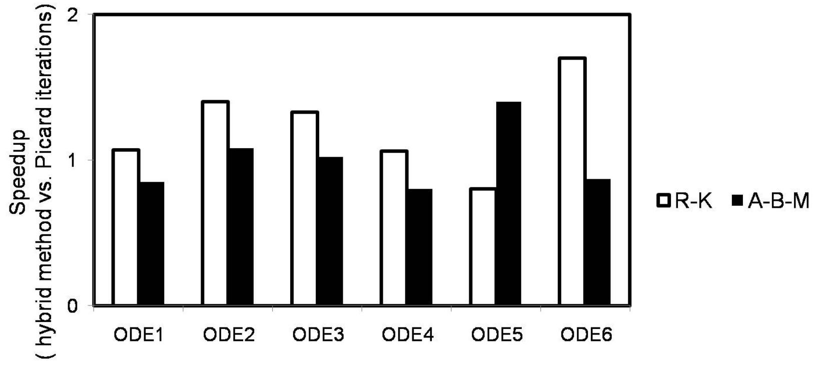


Figure 14: Relative speedup results, with global error in the order of 10^{-6}. The bars show the relative speedup results of the hybrid method divided against that of Picard iterations.

per time step between the quadrature computation (Simpson's rule) and the ODE solver (Runge-Kutta method and Adams-Bashforth-Moulton method). Fig. 15 shows the speedup of Picard iterations on 8 cores over two sequential ODE solvers with Runge-Kutta method and Adams-Bashforth-Moulton method respectively.

4. Related Work

Two other important dynamic iterations methods are the Waveform Jacobi method and the Waveform Gauss-Seidel method. The definitions of g for these are provided in [3].

The Waveform Jacobi recurrence is like that of equation (3), except that in the evaluation of the j th component of f, the j th component of u^m is replaced by the j th component of u^{m+1}. This decomposes the ODE with n variables into n ODEs where one variable is updated in each ODE. In the evaluation of f, the value of $u^m(s)$ is used for all other variables.

Similarly, the Waveform Gauss-Seidel recurrence is like that of equation (3), except that in the evaluation of the j th component of f, all components of u^m with index less than or equal to j are replaced by the corresponding components of u^{m+1}. This decomposes the ODE with n variables into n

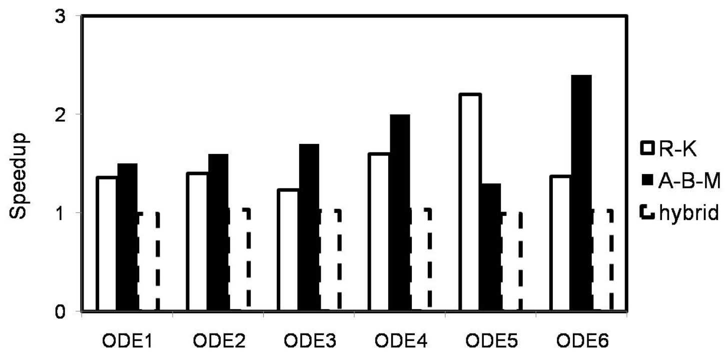


Figure 15: Speedup of Picard iterations over sequential ODE solvers on 8 cores.

ODEs where one variable is updated in each ODE. The ODE for the first component is solved as in Waveform Jacobi. The ODE for the second component uses the values of the first component of $u^{m+1}(s)$ evaluated before, and the values of $u^m(s)$ for the third and higher indexed components, in each evaluation of f. This process continues, with the ODE for each component using the values of $u^{m+1}(s)$ for components with smaller indices, and $u^m(s)$ for components with larger indices, in each evaluation of f.

Note that, despite the use of an ODE solver instead of quadrature by these methods, they are quite different from the hybrid-scheme. Their basic goal is to decouple the computation of each component of the system. They have the high memory requirement of the Picard iteration, because u^m needs to be stored for all time steps.

Each iteration of Waveform Jacobi can be parallelized by having the ODE for each component solved on a different core. This is advantageous for large systems, especially when different time steps can be used for different components. However, the communication cost can be high, because the components of u at all time points will need to be sent to other cores. Parallelizing it along the time domain is difficult in general. Some special cases can, however, be parallelized in time [3].

A wavefront approach is used to parallelize Waveform Gauss-Seidel [3]. The idea behind this is basically pipelining. Note that once the first time step of the first component of u^{m+1} has been completed, the first time step of the second component can be started, while the second time step of the first component is simultaneously started. If we need to perform m iterations with n time steps each, then the total of nm evaluations of f can be performed in $n + m$ steps in parallel. This appears to be a good reduction in time. However, we note that a conventional sequential ODE solver will solve the same problem in n time steps. Thus, even with parallelization, this method is not faster than an equivalent conventional solver[2].

The above processes of splitting the ODE into subcomponents can be generalized by keeping blocks of variables that are strongly coupled together. The blocks may also overlap [9]. When (2) is solved exactly, the solution converges superlinearly ([3, 10]) on a finite time interval. However, numerical methods have to be used to solve it in practice. Theoretical results on convergence for discrete version of dynamic iterations are presented in [11, 3]. Estimates of suitable window sizes for good convergence are provided in [12]. In realistic situations, this class of methods has often been found to converge slowly, which is a major limitation of these methods in practice.

Different strategies have been proposed to speed up the convergence of dynamic iterations. Coarse time step size for the earlier iterations, and finer time steps for later iterations, is proposed in [10]. A two-step-size scheme for solving Picard iterations is proposed in [7]. A fine step size is used for a fixed number of iterations in all the sub-systems to smooth the iterations. A coarse step size is then used to solve the residue of the solution restricted to the coarse time-mesh. Multigrid techniques have also been used for accelerating convergence ([13, 14]). Reduced order modeling techniques have been combined with dynamic iterations, in order to help propagate the change in one component of the system faster to the rest of the system in [15].

A different approach to time parallelization uses a coarse-grained model to guess the solution, followed by correction. The Parareal method can be considered an example of this ([16, 17]). Dynamic data-driven time parallelization ([18, 19, 20]) uses results from prior, related, runs to construct a coarse-grained model.

Conventional parallelization of ODEs is through distribution of the state

[2]However, this method may be useful for other reasons, such as when widely different time steps can be used for different components. But the order in which variables are solved has to be chosen carefully, to prevent the variable with small times steps from becoming a bottleneck to variables that depend on it [3].

space, as mentioned earlier. The difficulty with small systems is that the computation time per time step is small. For example, each step in our computation takes the order of a tenth of a microsecond using the Runge-Kutta method. This will be even smaller when the state space is distributed across the cores. On the other hand, MPI communication overhead is of the order of microseconds. One could use threads for better speed. However, the thread synchronization overhead is then significant.

5. Conclusion

We showed that the hybrid dynamic iterations yield significantly better performance than an equivalent sequential ODE solver. This is important because conventional parallelization in the spatial domain is not feasible for small ODE systems, as the communication overhead surpasses the computation time. In the hybrid method, the overall parallelization overheads are comparably smaller than the computation for the window size. The loss in the efficiency of the hybrid method is primarily due to the number of iterations required for convergence. The hybrid method can provide substantial benefit when effective, and it is no worse than an equivalent sequential solver when less effective.

We have considered three variants of the hybrid method: (i) solving for the entire time span; (ii) using fixed windows; (iii) using sliding windows. The former two have been studied for dynamic iterations. We demonstrated through the speedup results that the parallel efficiency of the sliding window method is better than the other two. We also showed that the convergence behavior of the hybrid method is better than Picard iterations. For higher accuracy solutions, the hybrid method achieves better speedup than Picard iterations. When lower accuracy solutions are acceptable, Picard iterations can have better performance than the hybrid method.

Acknowledgments

This work was funded by NSF grant #DMS 0626180. The authors would like to thank Xin Yuan at Florida State University for for permitting use of his Linux cluster. A.S. gratefully acknowledges his debt to Sri S. S. Baba, whose help and inspiration were crucial to the success of this work.

References

[1] Y. Yu, A. Srinivasan, Dynamic iterations for the solution of ordinary differential equations on multicore processors, in: Proceedings of the 23rd International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2009.

[2] E. Lelarasmee, A. E. Ruehli, A. L. Sangiovanni-Vincentelli, The waveform relaxation method for time-domain analysis of large scale integrated circuits, IEEE transactions on CAD IC systems 1 (1982) 131–145.

[3] C. W. Gear, Waveform methods for space and time parallelism, Journal of Computational and Applied Mathematics 38 (1991) 137–147.

[4] L. F. Shampine, Numerical Solution of Ordinary Differential Equations, Chapman & Hall, 1994.

[5] I. K. Youssef, H. A. El-Arabawy, Picard iteration algorithm combined with gauss-seidel technique for initial value problems, Applied Mathematics and Computation 190 (2007) 345–355.

[6] T. E. Hull, W. H. Enright, B. M. Fellen, A. E. Sedgwick, Comparing numerical methods for ordinary differential equations, SIAM Journal of Numerical Analysis 9 (1972) 603–737.

[7] B. Leimkuhler, Timestep acceleration of waveform relaxation, SIAM Journal on Numerical Analysis 35 (1998) 31–55.

[8] J. Sun, H. Grotstollen, Fast time-domain simulation by waveform relaxation methods, IEEE Transactions on Circuit and Systems 44 (1997) 660–666.

[9] R. Jeltsch, B. Pohl, Waveform relaxation with overlapping splittings, SIAM Journal on Scientific Computing 16 (1995) 40–49.

[10] O. Nevanlinna, Remarks on Picard Lindelöf iteration, BIT Numerical Mathematics 29 (1989) 328–346.

[11] M. Bjorhus, A note on the convergence of discretized dynamic iterations, BIT Numerical Mathematics 35 (1995) 291–296.

[12] B. Leimkuhler, Estimating waveform relaxation convergence, SIAM Journal on Scientific Computing 14 (1993) 872–889.

[13] C. Lubich, A. Ostermann, Multi-grid dynamic iterations for parabolic equations, BIT Numerical Mathematics 27 (1987) 216–234.

[14] S. Vandewalle, R. Piessens, On dynamic iteration methods for solving time-periodic differential equations, SIAM Journal on Numerical Analysis 30 (1993) 286–303.

[15] M. Rathinam, L. R. Petzold, Dynamic iteration using reduced order models: A method for simulation of large scale modular systems, SIAM Journal on Numerical Analysis 40 (2002) 1446–1474.

[16] L. Baffico, S. Bernard, Y. Maday, G. Turinici, G. Zerah, Parallel-in-time molecular-dynamics simulations, Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) 66 (2002) 57701–57704.

[17] Y. Maday, G. Turinici, Parallel in time algorithms for quantum control: Parareal time discretization scheme, International Journal of Quantum Chemistry 93 (2003) 223–238.

[18] A. Srinivasan, N. Chandra, Latency tolerance through parallelization of time in scientific applications, Parallel Computing 31 (2005) 777–796.

[19] A. Srinivasan, Y. Yu, N. Chandra, Application of reduced order modeling to time parallelization, in: Proceedings of HiPC 2005, Lecture Notes in Computer Science, volume 3769, Springer-Verlag, 2005, pp. 106–117.

[20] Y. Yu, A. Srinivasan, N. Chandra, Scalable time-parallelization of molecular dynamics simulations in nano mechanics, in: Proceedings of the 35 th International Conference on Parallel Processing (ICPP), IEEE, 2006, pp. 119–126.