

Dynamic Iterations for the Solution of Ordinary Differential Equations on Multicore Processors

Yanan Yu
Computer Science Department
Florida State University
Tallahassee FL 32306, USA
Email:yu@cs.fsu.edu

Ashok Srinivasan
Computer Science Department
Florida State University
Tallahassee FL 32306, USA
Email:asriniva@cs.fsu.edu

Abstract

In the past few years, there has been a trend of providing increased computing power through greater number of cores on a chip, rather than through higher clock speeds. In order to exploit the available computing power, applications need to be parallelized efficiently. We consider the solution of Ordinary Differential Equations (ODE) on multicore processors. Conventional parallelization strategies distribute the state space amongst the processors, and are efficient only when the state space of the ODE system is large. However, users of a desktop system with multicore processors may wish to solve small ODE systems. Dynamic iterations, parallelized along the time domain, appear promising for such applications. However, they have been of limited usefulness because of their slow convergence. They also have a high memory requirement when the number of time steps is large. We propose a hybrid method that combines conventional sequential ODE solvers with dynamic iterations. We show that it has better convergence and also requires less memory. Empirical results show a factor of two to four improvement in performance over an equivalent conventional solver on a single node. The significance of this paper lies in proposing a new method that can enable small ODE systems, possibly with long time spans, to be solved faster on multicore processors.

1. Introduction

Parallel processing is now widely available due to the popularity of multicore processors. A desktop, for example, may have two quad-core processors. It may also be equipped with a GPU having 100-200 cores. The Cell processor has eight cores called SPEs, which are meant to handle the computational workload. Two of these can be combined to form a 16-core shared memory machine. The Sony PS3 uses the same chip and makes six SPE cores available for use. Thus 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 does. However, 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. Applications will need to efficiently exploit the available parallelism, in order to get improved performance. This can be difficult for small problems, because of the finer granularity.

We will consider the specific problem of solving an Initial Value Problem involving an ordinary differential equation. We provide below a simple description of the problem and the conventional approach to solving it. (Readers with a background in ODEs can skip the next two paragraphs.) A first order ODE is given in the form $du/dt = f(t, u)$, where u , which may be a vector, is the state of a physical system, and t is often time. Higher order ODEs can be expressed as a first order ODE system using a standard transformation [13]. The initial state of the system at some time, $u(t_0)$, 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 will iteratively determine these values by starting with the known value of u at t_0 , and use $u(t_i)$ to find $u(t_{i+1})$. Conventional parallelization of this procedure distributes the components of u amongst different processors, and all processors update the values of u assigned to them, at each time step. In order to perform this update, each processor computes the components of f for the components of u assigned to it. They will need some components of u assigned to other processors, and thus communication is required. In a distributed memory computing environment, the communication cost is at least the order of a microsecond using a fast network such as Infiniband. In the shared memory programming paradigm on symmetric multiprocessor systems, the thread synchronization overhead is the order of a microsecond. Therefore, the computation involved on a processor should be large enough that the parallel overhead does not become a bottleneck. When the state space is small, the communication overhead dominates the computation, and parallelization is not feasible. Thus any improvement

over the sequential performance is of much benefit in such situations.

For small ODE systems, the solution time is significant when the number of time steps is large. Parallelization of the time domain appears promising then. Dynamic iterations are a class of methods that are often suitable for time parallelization. Here, we start out with an initial guess for the entire time domain, and then iteratively improve it until it converges. Picard iterations, which pre-date modern ODE solvers, are easy to parallelize along the time domain – each processor updates a different time interval independently, followed by a parallel prefix computation to correct the update. This method is explained further in §2. Dynamic iterations were well studied in the 1980s and 1990s. However, they were not suitable for most realistic problems due to their slow convergence. They also have a high memory requirement, because the solution at all time steps in the previous iteration is needed for the update in the next iteration.

In this paper, we propose a hybrid scheme that combines conventional sequential ODE solvers with dynamic iterations. We show that the convergence is faster with this scheme. This scheme also guarantees progress – in each iteration, at least one time interval converges. Thus, in the worst case, the total computation time is no worse than that of a sequential solver, apart from a small parallelization overhead. Of course, we hope to improve on a sequential solver in a typical case. We show later that we obtain significant improvement in performance on some sample ODE systems, compared with an equivalent conventional solver. We also compare it with Picard iterations. The hybrid method converges about twice as fast as Picard iterations.

The outline of the rest of this paper is as follows. In §2 we explain dynamic iterations, and Picard iterations in particular. We then describe the new approach in §3 and discuss its correctness and convergence. We demonstrate its effectiveness in §4. We discuss other time parallelization techniques in §5. We summarize our conclusions and present future work in §6.

2. Dynamic Iterations

The modern development of dynamic iterations for the solution of Initial Value Problems started with the work on Waveform Relaxation for the solution of large differential-algebraic equations in VLSI circuit simulations [7]. Many variants of the original method were then developed in the 1980s and 1990s. A unified definition of this class of methods is presented in [3]. It is also shown there that Picard iterations, which pre-date waveform relaxation by almost a century, can be considered a special case of this class of methods. We use the notation of [3] in the description below. Consider an initial value problem,

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

where $u \in \mathbb{R}^n$, $f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$, and $\dot{u} := du/dt$. Dynamic Iterations solve (1) recursively using the following equation.

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

The recursion is normally started using the initial guess $u^0(t) = u_0$. If g is defined such that $g(u, u) = 0$, then it can be shown that if the iteration converges, then it converges to the exact solution, provided (2) is solved exactly [3]. In practice, numerical methods are used to solve (2), and the approximations used affect the results, as shown in §4.

Different methods differ in their choice of g , which influences the convergence behavior and also the ease of parallelization. Picard iterations choose $g(y, z) = 0$. We discuss other common choices of g in §5. Dynamic iterations converge under fairly weak conditions [3], and so can be applied in most practical situations. The Picard method can also be parallelized efficiently, as explained below.

Picard iterations recursively solve (2) as follows. Initially, u^0 is stored at discrete time points. In the $(m+1)$ th iteration, $f(t, u^m)$ is computed at those time points. Then f is numerically integrated using some quadrature method. This leads to the following expression for determining u^{m+1} , when the integration is exact.

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

The above equation can be rewritten as follows.

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

Here, t_i is a sequence of time points. There may be several time points at which u^m are known, in the time interval $[t_i, t_{i+1}]$. The integral in (4) is evaluated using a quadrature rule.

The above method is parallelized easily. Each processor i , $0 \leq i < P$ is responsible for determining the values of u^{m+1} in the interval $(t_i, t_{i+1}]$, using the values of u^m . Each of them first evaluates the integrals in (4) for their time points using a quadrature method, for which they will need some boundary values of u^m available on neighboring processors. Then, a parallel prefix computation on the integral for t_i s is performed to compute the cumulative integrals from t_0 to t_i . After this, each processor can independently compute the values of u^{m+1} for which they are responsible. If the time interval (t_i, t_{i+1}) contains sufficient time points at which u are evaluated, then the computation cost will be much larger than the prefix cost, and thus the parallelization will be efficient.

While the parallel overhead is often small, the efficiency of parallelization of any dynamic iteration should be compared against an equivalent conventional ODE solver (that is, having the same error order as the quadrature), rather than against the dynamic iteration on one processor, in order to get an idea of the effectiveness of dynamic iterations.

Under this criteria, note that if the computations converge in N iterations, then the efficiency cannot be greater than $1/N$. For example, $N = 5$ will lead to relative efficiency less than 20%. Thus, dynamic iterations are suitable when conventional methods give poor parallel efficiency.

Dynamic iterations have generally not found widespread use because they were slow to converge, making them uncompetitive with conventional solvers. For example, fig. 1 shows four iterations being needed for convergence (just from a visual perspective) for a simple equation. However, we believe that with the popularity of multicore processors, it is time to re-evaluate the usefulness of dynamic iterations. As mentioned earlier, we now have a situation where users can solve small problems in parallel. Conventional solvers are not effective for this, and so any gain in speedup is a benefit, even if the efficiency is low. For example, an efficiency of 20% on eight cores gives a speedup of 1.6 over a conventional solver.

3. A Hybrid Scheme

We will present our method as an intuitive improvement to the Picard method to deal with one of the latter's shortcoming, and then we will analyze its convergence properties.

Apart from slow convergence, another problem with the Picard method, of somewhat less importance, is its large memory requirement. In order to simulate n time steps, the value of u^m needs to be stored for the n time points. Usually one divides the total time span into smaller windows, and waits for convergence of the entire window before moving to the next window. However, the number of time steps in each window can still be high. This can affect performance through increased cache misses.

In the proposed method, only the results at certain time points t_i are stored. The time steps are usually small for stability and accuracy reasons. However, because the result of the exact solution at each time step is not always required, we need not to store the results at all intermediate time steps, whereas Picard iterations need the intermediate values inside each interval $[t_i, t_{i+1}]$. Of course, the granularity of the t_i s in the hybrid method should be at least as fine as the resolution desired by the user.

We first note that the exact solution of eqn. (1) is given by the following.

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

If we compare with the Picard recurrence, eqn. (4), we see that Picard iterations replace u with u^m in the integral, because that is the latest approximation to u available to it. In our method, we have u^m available only at the points t_i . In our method, processor i will solve the exact ODE eqn. (1) with initial condition $\hat{u}_{(i)}(t_i) = u_i^m$, up to time t_{i+1} . The hat on u indicates that this is a solution to the exact ODE

with a possibly wrong initial condition, and the subscript (i) indicates that this problem is solved on processor i . Note that $\hat{u}_{(i)}(t_{i+1}) = \hat{u}_{(i)}(t_i) + \int_{t_i}^{t_{i+1}} f(s, \hat{u}_{(i)}(s)) ds$. Thus we get the following expression.

$$\hat{u}_{(i)}(t_{i+1}) - u_i^m = \int_{t_i}^{t_{i+1}} f(s, \hat{u}_{(i)}(s)) ds, \quad (6)$$

where we have used the initial condition $\hat{u}_{(i)}(t_i) = u_i^m$. We replace the integral in the Picard iterations recurrence, eqn. (4) with the integral above, and this integral, in turn, can be replaced by the left hand side of the expression above to give the following recurrence for the hybrid method.

$$u_{i+1}^{m+1} := u_i^{m+1} + \hat{u}_{(i)}(t_{i+1}) - u_i^m. \quad (7)$$

The parallelization is similar to that of the Picard method, and is shown in Algorithm 3.1. The $\hat{u}_{(i)}(t_{i+1})$ are computed by solving an ODE, independently on each processor. Then a parallel prefix computation is performed as in the Picard method and the new approximation u_{i+1}^{m+1} is computed. An intuitive way of thinking about this method is as follows. If u_i^m were accurate, then $\hat{u}_{(i)}(t_{i+1})$ would be the exact solution for time t_{i+1} . However, we observe an error in the value of $u_i^{m+1} - u_i^m$ between two successive iterations at time t_i . We add this as a correction factor to $\hat{u}_{(i)}(t_{i+1})$.

Algorithm 3.1: HYBRID-METHOD($u_i^0 = u_0$,
Number of processors P)

repeat

do $\left\{ \begin{array}{l} \text{for each processor } i \in \{0..P-1\} \\ \text{do } \left\{ \begin{array}{l} \hat{u}_{(i)}^m(t_{i+1}) \leftarrow \text{Solve ODE} \\ \quad \text{(with initial condition } u_i^m) \\ \Delta_i \leftarrow \hat{u}_{(i)}^m(t_{i+1}) - u_i^m \\ G_i \leftarrow \text{Parallel prefix on } \Delta_i \\ u_{i+1}^{m+1} \leftarrow G_i + u_0 \end{array} \right. \end{array} \right.$

until Convergence

We next show that if the iterations converge, then they converge to the exact solution, provided the ODE is solved exactly in each iteration. Of course, the ODE is actually solved numerically, with approximation errors. We evaluate the effect of these in §4.

Assume that the iterations converge. At convergence, $u_i^{m+1} = u_i^m$, for $i \in [1, n]$. From eqn. (7), we get $u_{i+1}^{m+1} = \hat{u}_{(i)}(t_{i+1})$. Processor 0 starts from the exact initial condition, and thus $u_1^{m+1} = \hat{u}_0(t_1)$ is exact. But $u_1^{m+1} = u_1^m$ due to convergence, and thus u_1^m is exact. Thus process 1 started from the exact solution in determining $\hat{u}_1(t_2)$ and thus $u_2^{m+1} = \hat{u}_1(t_2)$ is exact. We can proceed in this manner, using an inductive argument to show that u^{m+1} is exact.

We now consider the issue of how fast the method converges. We first explain why we intuitively expect it to have better convergence than Picard iterations. We then prove that it converges in a finite number of steps, if the ODE solution is exact. We will demonstrate in §4.2 that it has good convergence properties in practice.

We saw above that the difference between Picard iterations and the hybrid method is in how the term $\int_{t_i}^{t_{i+1}} f(s, u(s)) ds$ is estimated, without knowing the true solution u . Picard iterations perform a quadrature using the approximation u^m . The hybrid method starts with a point on u^m , but uses the exact differential equation. Thus, we can expect its solution to be better than that of Picard iterations, which use the old approximation throughout the time domain.

We also show that the hybrid method always progresses. That is, in each step, at least one additional time interval converges. Note that processor 0 starts with the exact initial condition and thus converges after the first iteration. In the second iteration, the initial condition for processor 1 is exact, and so its solution u_2^2 is exact. Using an inductive argument, u_i^i is exact. Thus the method converges in n iterations if there are n time intervals. Of course, this would not give any speedup over an equivalent sequential solver, and we would hope to do better in practice. However, it does provide a guarantee that in the worst case, the performance will not be worse than an equivalent conventional sequential solver, if the parallelization overheads are small. Note that the memory consumption of this solver on each processor is small – roughly the same as a sequential solver. It is independent of the number of time steps in a time interval, unlike dynamic iterations.

We consider three variants of this method. (i) We solve for the entire time domain by dividing the time span T into T/P intervals on P processors. (ii) We use windows of a certain width W , with the number of windows being much larger than P . The processors wait for convergence of the first P windows (one window per processor) before proceeding with the next set of P windows. This type of windowing has been shown by others to be more effective than solving for the entire time domain in dynamic iterations [16]. (iii) We notice that after the first few processors have converged, they don't perform useful work by repeating the calculations. So, these processors will be used for computing new windows. We call this method *sliding windows*. Note that a processor cannot be considered as having converged until it has locally converged *and* all the processors handling smaller values of time have also converged.

4. Empirical Evaluation

4.1. Experimental Setup

The experimental tests were performed on an Intel Xeon cluster. Each node has two 2.33 GHz Quad-core CPUs with 8 GB shared memory. OpenMPI 1.2 is used for inter-process communications. Both the sequential and parallel code are compiled with optimization level $-O3$. All the speedup and relevant results are obtained on 8 cores on a single node.

We use an assembly timing routine, which accesses the time-stamp counter through the *rdtsc* instruction, for our timing results. The resolution of the timer is $0.05\mu s$. It is sufficiently good for the timing purpose in this paper, where the timing results are at least the order of a few microseconds.

The results are presented by solving three ODE systems. They are: (i) ODE1: $u'' = -\omega^2 u$, with $\omega = 1$, $u(0) = 0$, $u'(0) = 1$ for $t \in [0, 6.4]$; (ii) ODE2 – Airy equation: $u'' = -k^2 ut$, with $k = 1$, $u(-6) = 1$, $u'(0) = 1$ for $t \in [-6, 2]$; (iii) ODE3 – Duffing equation: $u'' + \delta u' + (\beta u^3 + \omega_0^2 u) = \gamma \cos(\omega t + \phi)$, with $\gamma = 2.3$, $\phi = 0$, $\delta = 0.1$, $\beta = 0.25$, $\omega_0 = 1$, $u(-6) = 1$, $u'(0) = 1$ for $t \in [-6, 2]$. The Airy equation is commonly seen in physics and astronomy. The Duffing equation describes the motion of a damped oscillator.

In this paper, a single step 4th order Runge-Kutta Method is implemented as the ODE solver used in the hybrid method. We use the Simpson's rule to evaluate the quadrature in the Picard method. The accuracy of the Simpson's rule is also order 4. All computations were in double precision.

The exact solutions to each of the three ODEs are shown in fig. 2, 3, 4. The exact solution to the first ODE can be expressed in the simple symbolic form, $y = \sin(\omega t)$. However, there are no closed-form solutions to the second and third ODEs in the real domain. In order to get good approximations to the exact solutions, we used the following procedure. We solved the ODEs using a sequential ODE solver. We then solved them again with a time step size that was an order of magnitude smaller, and determined the largest difference in values at any given time point. We repeated this with smaller time steps, until this largest difference was the order of 10^{-10} . This solution was then used as an approximation to the exact solution. In §4.2, we chose the time step size such that the global error is of the order of 10^{-6} . Therefore, the accuracy in the approximated exact solution is sufficient for the comparisons in this paper.

In the results below, we use the terms *tolerance* and *error*. The latter refers to the absolute difference between the exact solution and the computed solution. The tolerance is a parameter to the ODE solver. When using dynamic iterations, the solution is considered to have converged if the absolute difference between solutions at successive iterations is smaller than this value. When used with a sequential ODE

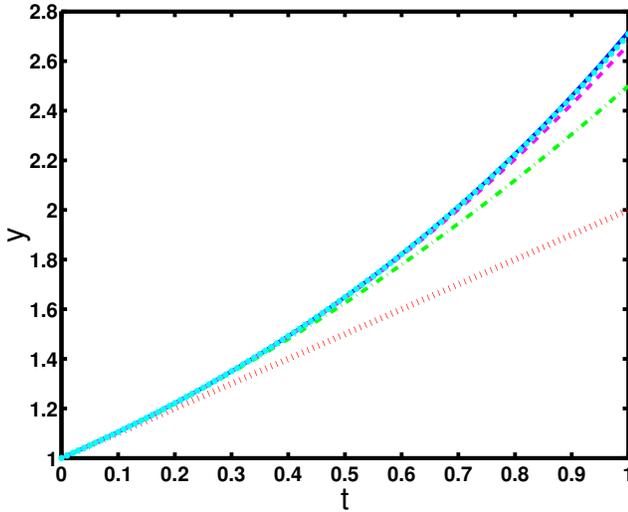


Figure 1. Convergence of Picard iteration for the solution of $\dot{u} = u, u_0 = 1$.

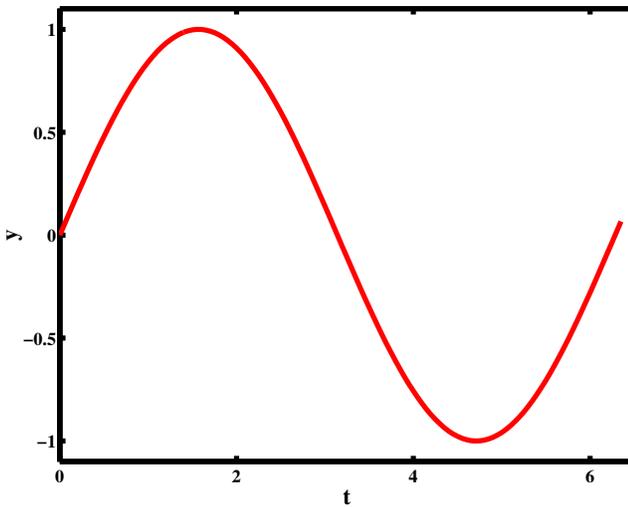


Figure 2. ODE1: $y'' = -\omega^2 y$

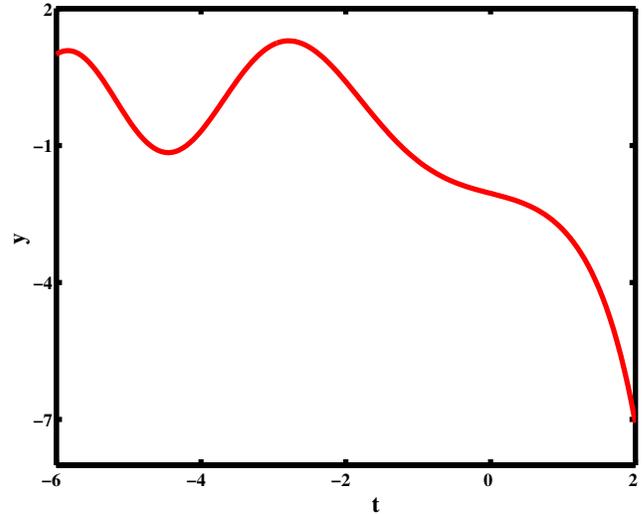


Figure 3. ODE2: $y'' = -k^2 y t$

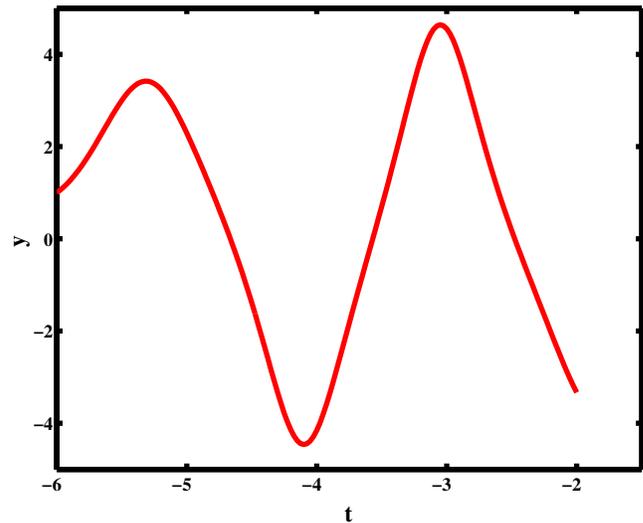


Figure 4. ODE3: $y'' + \delta y' + (\beta y^3 + \omega_0^2) = \gamma \cos(\omega t + \phi)$

solver, the tolerance has the conventional meaning. The error may be larger than the tolerance.

The speedup results compare the time of dynamic iterations with the time for a sequential 4th order Runge-Kutta Method. Speedup is close to linear when we compare with a sequential implementation of dynamic iterations. However, in order to realistically evaluate the usefulness of dynamic iterations, we need to compare them against a conventional 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.

4.2. Experimental Results

We first explain our choice of time step size. If the time step size is smaller than necessary, then the granularity of the computation is coarser than necessary, and thus the communication overheads are artificially made small, relative to the computational effort in its parallel implementation. In order to avoid this, we first evaluate the accuracy of the sequential ODE solver for different time step sizes, and choose one that gives good accuracy. For stability, a step size of 0.005 is sufficient. However, a much smaller step size is usually required for accuracy. Fig. 5 shows the comparison of the errors from *ode45* function in Matlab (with parameters of absolute error = 10^{-6} and relative error = 10^{-6}) and *dlsode* function in ODEPACK. We need a step size of 10^{-7} for the

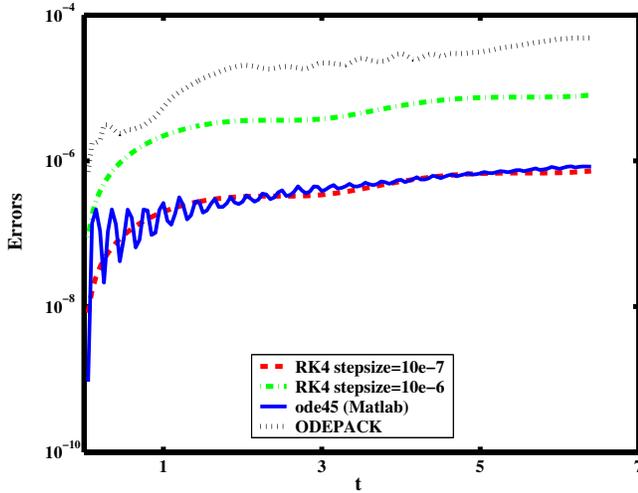


Figure 5. Error versus time for different values of stepsize h using the sequential solver for ODE1. Errors are compared with that of ode45(Matlab) and dlsode(ODEPACK).

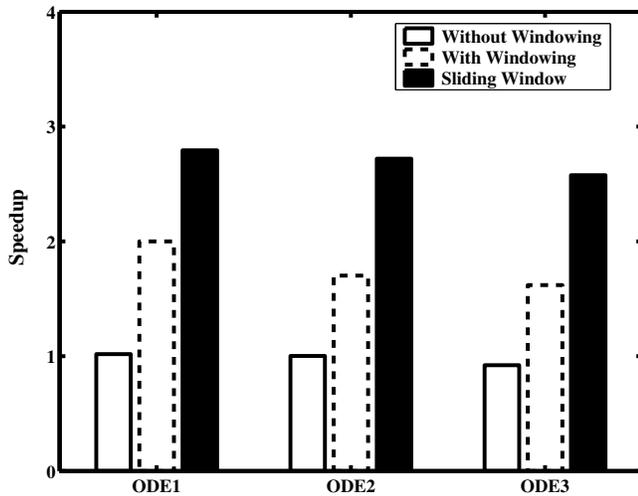


Figure 6. Speedup result of the three ODEs using different variants of the hybrid method. The window size is 10^5 time steps and the tolerance is 10^{-6} .

sequential ODE solver to have the same order of accuracy as *ode45* in Matlab when solving ODE1. Similar experiments were performed for ODE2 and ODE3 and their step sizes were also determined as 10^{-7} .

As mentioned in §3, the hybrid method can have three different variants – without windowing, with windowing, and with sliding windows. We wish to compare the effectiveness of each of these variants. Fig. 6 shows the speedup results using the three variants on the ODEs. The window size is 10^5 time steps for both the windowing methods. We can see that sliding windows yields a better speedup than fixed

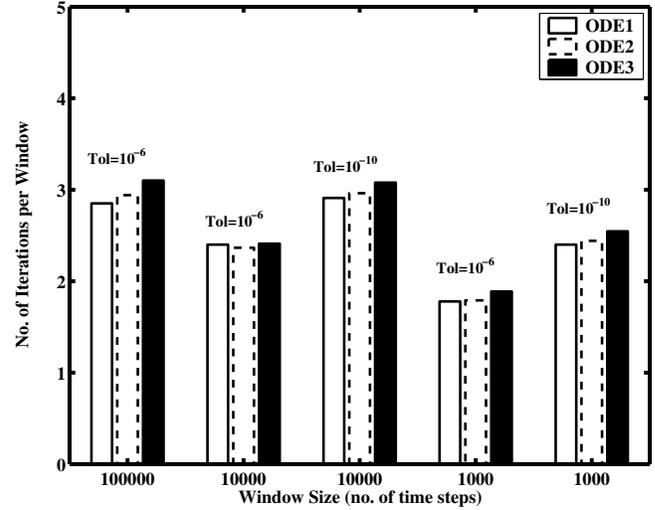


Figure 7. Average number of iterations using different window sizes and tolerances with sliding windows.

windows. The non-window method does not perform well for such small numbers of processors, because it requires more than eight iterations, on eight cores, to converge for the given time span. Given these observations, we will only use results from the sliding windows method for comparisons with the sequential solver as well as with Picard iterations, in the following discussions.

The hybrid method with sliding windows requires three MPI function calls per iteration. *MPI_Scan* is used for the parallel prefix summation, *MPI_Allreduce* is called in testing for convergence, and *MPI_Isend/MPI_Recv* is used to transfer the latest updated value between certain processors. Across 8 cores on the same node, *MPI_Scan* and *MPI_Isend/MPI_Recv* take less than $1\mu s$ and *MPI_Allreduce* takes less than $10\mu s$. The computation time per step for the three sample ODEs are $0.10\mu s$, $0.11\mu s$ and $0.44\mu s$. For speedup results shown in the following discussions, the window size is at least 10^3 time steps. In that case, the granularity of the computation is of the order of a hundred microseconds, and the overall communication overhead at most the order of 10%.

We now discuss the choice of window size. As mentioned before, the window size affects the granularity of the parallelism. For exact ODE solvers, smaller granularity indicates relatively larger communication overhead. Thus the speedup can be expected to decrease with decrease in window size. In the hybrid method, the window size can also affect the number of iterations as well as the numerical accuracy of the solution. We experimented with three different window sizes, 10^5 time steps, 10^4 time steps and 10^3 time steps. Fig. 7 compares the average number of iterations per window for the solutions to converge. As we can see, if we use the same order of error tolerance for all the window sizes, the

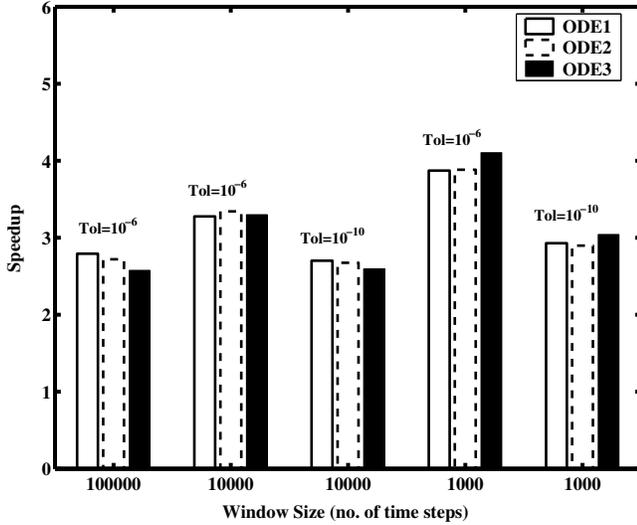


Figure 8. Speedup results using different window sizes and tolerances with sliding windows.

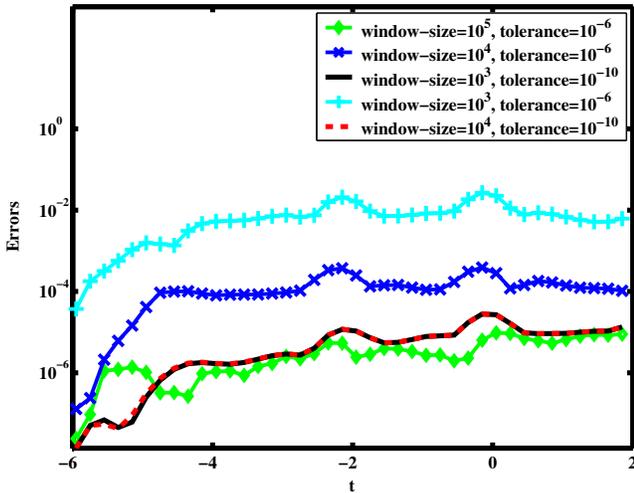


Figure 9. Accuracy of the solution using different window sizes and tolerances for ODE3. (The black solid line almost overlaps with the red dashed line.)

smaller the window size is, the faster the method converges. Fig. 8 compares the speedup when different window sizes and tolerance are used to solve the three ODEs. However there is a trade-off with the accuracy of the solution. Fig. 9 shows how the accuracy is affected by different window sizes and local error tolerances. The error is about two orders larger when the window size is decreased by one order of magnitude, when the tolerance is 10^{-6} . By comparing the effect of different window sizes and tolerances on both speedup and accuracy, we chose the window size of 10^3 time steps. Similar conclusions were also drawn for ODE1 and ODE2.

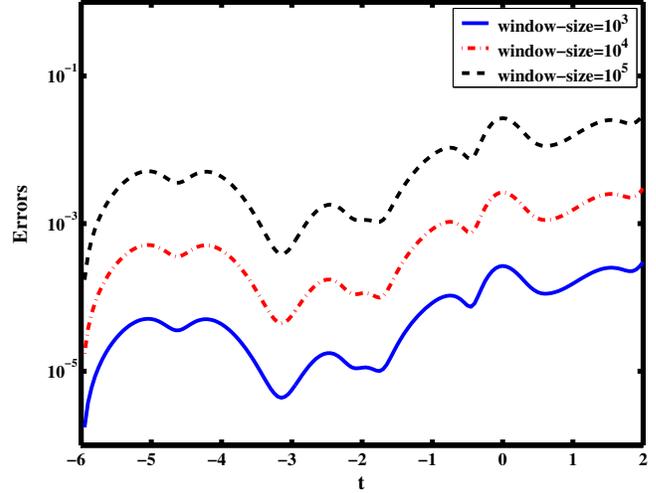


Figure 10. Errors using different window sizes in the Picard method with tolerance equal to 10^{-10} to solve for ODE3.

The window size also affects the accuracy and the convergence rate of Picard iterations. Fig. 10 shows the error versus the window size for ODE3. The step size used for the quadrature integration is the same as that of the ODE solver for our modified method. The solid line uses the same window size of 10^3 and tolerance of 10^{-10} . As we can see the accuracy is about 1 order lower than that of our method. Fig. 11 shows the comparison of the number of iterations of the Picard method to the hybrid method for the equivalent parameters of window size, stepsize and tolerance. The y-axis is the ratio of the number of iterations required by Picard iterations versus the number of iterations required by the hybrid method. The convergence rate is slower in Picard iterations in all the three cases. However, note that this improvement in convergence rate does not directly translate to a proportional improvement in speedup, because the Runge-Kutta method evaluates the ODE at 5 points for each time step, while the Picard iteration, using Simpson's rule, only calls the function f in equation (1) once for each time step.

5. Related Work

We discussed dynamic iterations using the Picard method in §2. Other methods can be derived by a different choice of g in equation (2). Two important cases 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

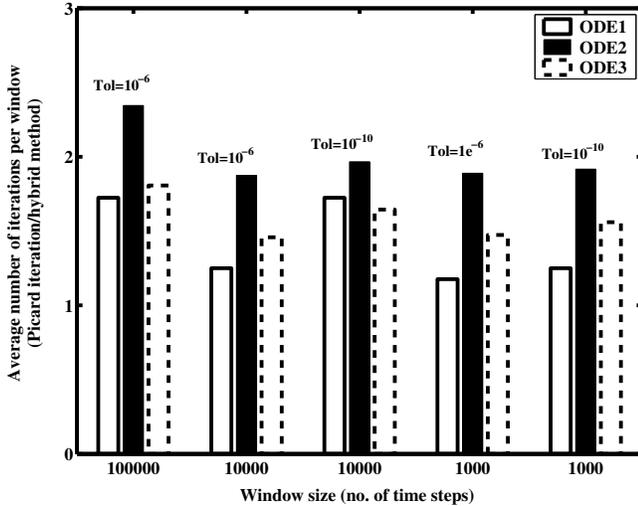


Figure 11. Ratio of the average number of iterations. Picard method vs. the hybrid method compared for the same window size and tolerance.

ODE. In the evaluation of f , the value of $u^m(s)$ is used for all other variables.

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 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 high memory requirement as that of Picard iterations, 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 processor. 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 processors. 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 d variables for n time steps, then the $m \times n \times d$ evaluations of f are performed in parallel in $d + m \times n$ time, if $d \leq n$. This method appears to parallelize well. However, note that a sequential ODE solver can solve the same problem with n evaluations of f . Thus, even with parallelization, this method is not faster than an equivalent conventional solver¹.

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 [4].

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 [2], [3]. Estimates of suitable window sizes for good convergence are provided in [5]. 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 [6]. 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 [8], [17]. 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 [11].

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 [1], [9]. Dynamic data-driven time parallelization [14], [15], [18] uses results from prior, related, runs to construct a coarse-grained model.

Conventional parallelization of ODEs is through distribution of the state 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

1. 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 time steps from becoming a bottleneck to variables that depend on it [3].

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 also use threads on multicore processors. However, the thread synchronization overhead is then significant, as mentioned earlier. In fact, a parallel OpenMP implementation took almost an order of magnitude more time than a sequential ODE solver, for the three ODEs considered in this paper.

6. Conclusions

We have shown that hybrid dynamic iterations yield significantly better performance than an equivalent sequential method. This is important because conventional parallelization is not feasible for small ODEs. The loss in efficiency in the hybrid scheme is primarily due to the number of iterations required for convergence. The other parallelization overheads are small. Thus, they provide substantial benefit when effective, and are not much worse than the sequential computation when less effective.

We have considered three variants of this method: (i) solving for the entire interval, (ii) using windows, and (iii) using sliding windows. The former two have been studied by others for dynamic iterations (see [5], [12], [16] for windowing). We demonstrated through the speedup results that the parallel efficiency of the third is better. We also showed that the convergence behavior of the hybrid method is better than Picard iterations. Furthermore, our method has smaller memory requirements than dynamic iterations. This can lead to better cache performance.

This paper has demonstrated the potential of hybrid dynamic iterations in providing a scalable solution for small ODE systems on modern computing platforms. We propose the following future work to further establish this technique. We are currently working on completing the following simple extensions of this paper. (i) We are evaluating more ODEs. (ii) We are implementing this approach on GPUs (including double precision) and on the Cell. The hybrid method's smaller memory requirement is especially useful on these platforms because of the small local store on the Cell and the small set of registers and shared memory on the GPU. (iii) We plan to replace the Runge-Kutta method, which requires multiple evaluations of f at each time step, by a multi-step method, which requires only one function evaluation at each step. Our plans for the future are as follows. (i) We wish to use an adaptive ODE solver instead of the fixed time-step Runge-Kutta method that we have currently used. This may require load balancing, when the time steps sizes on different processors are different. (ii) Our error tolerance condition is currently very simple, and just based on the absolute difference between different iterations. Conventional solvers use error estimates to determine the tolerance. We need to develop theoretical results to enable us

to estimate the error, based on the time steps size, tolerance, and window width. We are also planning a Matlab interface so that the Matlab `ode45` solver can be replaced by our solver when a GPU is available.

Acknowledgments

This work was funded by NSF grant # DMS 0626180. We also thank Xin Yuan at Florida State University for permitting use of his Linux cluster. A.S. expresses his gratitude to Sri Sathya Sai Baba for his help and inspiration.

References

- [1] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zerah. Parallel-in-time molecular-dynamics simulations. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 66:57701–57704, 2002.
- [2] M. Bjorhus. A note on the convergence of discretized dynamic iterations. *BIT Numerical Mathematics*, 35:291–296, 1995.
- [3] C. W. Gear. Waveform methods for space and time parallelism. *Journal of Computational and Applied Mathematics*, 38:137–147, 1991.
- [4] R. Jeltsch and B. Pohl. Waveform relaxation with overlapping splittings. *SIAM Journal on Scientific Computing*, 16:40–49, 1995.
- [5] B. Leimkuhler. Estimating waveform relaxation convergence. *SIAM Journal on Scientific Computing*, 14:872–889, 1993.
- [6] B. Leimkuhler. Timestep acceleration of waveform relaxation. *SIAM Journal on Numerical Analysis*, 35:31–50, 1998.
- [7] E. Lelarsmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli. The waveform relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 1:131–145, 1982.
- [8] Ch. Lubich and A. Ostermann. Multi-grid dynamic iterations for parabolic equations. *BIT Numerical Mathematics*, 27:216–234, 1987.
- [9] Y. Maday and G. Turinici. Parallel in time algorithms for quantum control: Parareal time discretization scheme. *International Journal of Quantum Chemistry*, 93:223–238, 2003.
- [10] O. Nevanlinna. Remarks on Picard Lindelöf iteration. *BIT Numerical Mathematics*, 29:328–346, 1989.
- [11] M. Rathinam and L. R. Petzold. Dynamic iteration using reduced order models: A method for simulation of large scale modular systems. *SIAM Journal on Numerical Analysis*, 40:1446–1474, 2002.
- [12] J. Sand and K. Burrage. A Jacobi waveform relaxation method for ODEs. *SIAM Journal on Scientific Computing*, 20:534–552, 1998.

- [13] L. F. Shampine. *Numerical Solution of Ordinary Differential Equations*. Chapman & Hall, 1994.
- [14] A. Srinivasan and N. Chandra. Latency tolerance through parallelization of time in scientific applications. *Parallel Computing*, 31:777–796, 2005.
- [15] A. Srinivasan, Y. Yu, and N. Chandra. Application of reduced order modeling to time parallelization. In *Proceedings of HiPC 2005, Lecture Notes in Computer Science*, volume 3769, pages 106–117. Springer-Verlag, 2005.
- [16] J. Sun and H. Grotstollen. Fast time-domain simulation by waveform relaxation methods. *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications*, 44:660–666, 1997.
- [17] S. Vandewalle and R. Piessens. On dynamic iteration methods for solving time-periodic differential equations. *SIAM Journal on Numerical Analysis*, 30:286–303, 1993.
- [18] Y. Yu, A. Srinivasan, and N. Chandra. Scalable time-parallelization of molecular dynamics simulations in nano mechanics. In *Proceedings of the 35 th International Conference on Parallel Processing (ICPP)*, pages 119–126. IEEE, 2006.