Parallel Implementation of a Validated Method for Initial Value Problems

Yu Wu
Department of Computing and Software
McMaster University
Hamilton, ON, Canada L8S 4L7

Sanzheng Qiao
Department of Computing and Software
McMaster University
Hamilton, ON, Canada L8S 4L7

Abstract ***Validated (interval) method is used to solve the initial value problems (IVPs) of ordinary differential equations (ODEs). Compare to traditional methods, validated method can provide a rigorous solution for IVPs of ODEs. But validated method requires more computational work than the traditional methods. We present a parallel implementation of the validated method and discuss our design idea, synchronization issues in this paper.***

Keywords: Validated methods, Interval methods, Ordinary differential equations, Initial value problems, Multithreading.

1 Introduction

We consider the problem of solving an Initial Value Problem (IVP) of an nth order Ordinary Differential Equation (ODE)

$$\frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \cdots, \frac{d^{n-1}y}{dt^{n-1}}\right)$$

with the following n initial conditions at t_0:

$$\begin{aligned} y\,(t_0) &= c_0, \\ \frac{dy}{dt}\,(t_0) &= c_1, \\ &\vdots \\ \frac{d^{n-1}y}{dt^{n-1}}\,(t_0) &= c_{n-1}, \end{aligned}$$

where $c_0, c_1, \cdots, c_{n-1}$ are given constants and t_0 can be a scalar or an interval.

An nth order ordinary differential equation can be transformed into a system of n first order ordinary differential equations. Letting

$$y_1 = y, \text{ and } y_i = \frac{d^{i-1}y}{dt^{i-1}} \text{ for } i = 2, ..., n,$$

we have the system

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}' = \begin{bmatrix} y_2 \\ y_3 \\ \vdots \\ y_n \\ f(t, y_1, \cdots, y_n) \end{bmatrix}$$

with the initial condition

$$\begin{bmatrix} y_1(t_0) \\ \vdots \\ y_n(t_0) \end{bmatrix} = \begin{bmatrix} c_0 \\ \vdots \\ c_{n-1} \end{bmatrix}.$$

Using vector notations, we write

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}), \text{ with } \mathbf{y}(t_0) = \mathbf{c},$$

where $\mathbf{y} = [y_1, ..., y_n]^{\mathrm{T}}$, $\mathbf{f}$ is a vector of functions, and $\mathbf{c} = [c_0, ..., c_{n-1}]^{\mathrm{T}}$.

This paper is organized as follows. We describe the Taylor series method for IVPs for ODEs in §2 and the validated methods in §3. The design ideas of our parallel implementation is presented in §4.

2 Taylor Series Method

For simplicity, we first consider a single first order ODE problem

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0.$$

Expanding the solution $y(t)$ at t_0, we have

$$y(t_1) = y(t_0) + \sum_{i=1}^{k-1} \frac{y^{(i)}(t_0)}{i!} h^i + \frac{y^{(k)}(t_0)}{k!} h^k + \cdots$$

where $h = t_1 - t_0 > 0$ is the step size. Let

$$y_1 = y_0 + \sum_{i=1}^{k-1} \frac{y^{(i)}(t_0)}{i!} h^i$$

be an approximation of $y(t_1)$, then we can use y_1 as the initial condition of the ODE $y' = f(t, y)$ to compute the next approximation $y_2 \approx y(t_1 + h)$. In general,

$$y_{j+1} = y_j + \sum_{i=1}^{k-1} \frac{y^{(i)}(t_j)}{i!} h^i. \qquad (1)$$

Thus, the Taylor series method is a single step explicit method.

Obviously, the key to the Taylor series method is the computation of the Taylor coefficients. We denote the ith term coefficient $y^{(i)}(t_j)/i!$ by $y_{(i)}$, then $y_{(0)} = y_j$. Since $y(t)$ is the solution of the ODE problem

$$y'(t) = f(t, y(t)), \quad y(t_j) = y_j, \qquad (2)$$

the first term coefficient

$$y_{(1)} = y'(t_j) = f(t_j, y_j) = f(t_j, y_{(0)}).$$

This shows that $y_{(1)}$ can be computed by $y_{(0)}$ using the function $f(t_j, y_{(0)})$. To establish a general relation between $y_{(i)}$ and the function $f(t, y)$, we consider the Taylor expansion of $f(t, y)$ at t_j

$$f(t_{j+1}, y_{j+1}) = f(t_j, y_j) + \sum_{i=1}^{k-1} \frac{f^{(i)}(t_j, y_j)}{i!} h^i.$$

Similarly, we denote its ith term coefficient $f^{(i)}(t_j, y_j)/i!$ by $f_{(i)}$, then $f_{(0)} = y_{(1)}$. Since $y(t)$ is the solution of the ODE problem (2), we have the following relation

$$y_{(i)} = \frac{y^{(i)}(t_j)}{i!} = \frac{f^{(i-1)}(t_j, y_j)}{i!} = \frac{1}{i} f_{(i-1)}. \quad (3)$$

The Taylor coefficient $f_{(i-1)}$ can be obtained by $y_{(0)}, ..., y_{(i-1)}$ using the Taylor coefficients of sums, products, quotients, and the standard functions. The overall recursive scheme is:

$$y_{(0)} \to f_{(0)} \to y_{(1)} \to \cdots .$$

In the case of a system of n first order ODEs, we set two matrices

$$T_{\text{in}} = \begin{bmatrix} y_{1(0)} & y_{1(1)} & \cdots & y_{1(k)} \\ y_{2(0)} & y_{2(1)} & \cdots & y_{2(k)} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n(0)} & y_{n(1)} & \cdots & y_{n(k)} \end{bmatrix} \qquad (4)$$

and

$$T_{\text{out}} = \begin{bmatrix} f_{1(0)} & f_{1(1)} & \cdots & f_{1(k-1)} \\ f_{2(0)} & f_{2(1)} & \cdots & f_{2(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ f_{n(0)} & f_{n(1)} & \cdots & f_{n(k-1)} \end{bmatrix}. \qquad (5)$$

Starting by the first column of T_{in}, we can compute the first column of T_{out}, which leads to the second column of T_{in} using the relation (3). Thus all the Taylor coefficients, the entries of T_{in}, can be obtained recursively column by column.

We use the software package TADIFF [3] to evaluate the Taylor coefficients. Figure 1 shows the part of the program which generates the Taylor coefficients.

```
while not finish
  Initialize Tin[0:n-1][0];
  for col = 0 to k-1
    for row = 0 to n-1
      Tout[row].eval(col);
      Tin[row][col+1] =
        Tout[row][col]/(col+1);
    end
  end
end.
```

Figure 1: Taylor coefficient generation

3 Validated Method for IVPs for ODEs

Standard numerical methods for solving the IVPs attempt to compute an approximate solution that satisfies a user-specified tolerance. Those general methods provide a robust and reliable way for most ODE applications. However, it is possible to find examples for which they return inaccurate results. When the calculations become critical of reliability and safety in a system, the general methods may not suit for solving these problems.

Comparing with the general methods, a validated method based on Taylor coefficients provides an guaranteed solution to contain a mathematically correct result. In the past, the validated methods have not been popular because they require much more storage and computational work than general methods for the same problem. Also, it is difficult to develop a robust software for validated methods in languages such as FORTRAN and C. Nowadays, the speed of processors is increasing, and the cost of chip is decreasing. Moreover, the Object-Oriented languages also provide a new flexible solution to develop software. A package, Validated Numerical Ordinary Differential Equation (VNODE) [1, 2] realizes the validate method for IVPs of ODEs in C++.

Different from the traditional methods, a validated method provides IVPs with intervals instead of constant real numbers. The scalar expansion (1) is extended to the interval form

$$[y_{j+1}] = [y_j] + \sum_{i=1}^{k-1} h^i f^{[i]}([y_j]) + h^k f^{[k]}([y_j]),$$

where $[y_j]$ denotes an interval and $f^{[i]}([y_j])$ denotes the ith term Taylor coefficient in interval form. The operator overloading of C++ can redefine all mathematical operations so that we can operate on intervals as real numbers.

The validated method for the IVPs of ODEs has two important advantages. First, if the method returns a solution, then the problem is guaranteed to have a unique solution. Second, an enclosure of the true solution is produced if the method returns successfully.

A validated method computes intervals $[y_j]$, $j = 1, 2, ..., m$, which are guaranteed to contain the solution $y(t)$ at $t_1, t_2, ..., t_m$. That is

$$y(t_j) \subseteq [y_j] \quad \text{for} \quad j = 1, 2, ..., m.$$

For each step, most Taylor series methods consist of two phases:

- Compute an *a priori* enclosure $[y_j]$ of the solution and select a stepsize h by using the Picard-Lindelöf operator and the Banach fixed-point theorem.

- Since the *a priori* enclosure will become wider and wider during the computation with the *wrapping effect* [5], we need to compute a tighter enclosure $[y_{j+1}]$ for $y(t_{j+1})$. Otherwise, the *a priori* enclosure will be too conservative to be meaningful.

Both phases use interval arithmetic to compute the Taylor coefficient matrices T_{in} (4) and T_{out} (5). The second phase, which computes a tighter enclosure, generates the Jacobian of Taylor coefficients.

4 Parallel Implementation of Validated Method

In this paper we consider a parallel implementation of the VNODE (Validated Numerical Ordinary Differential Equations) method [1, 2]. This method is based on high-order Taylor series expansions. The coefficients of Taylor series are generated by automatic differentiation methods [4] such as TADIFF [3]. Specifically, the Taylor coefficients are generated recursively as described in §2. It can be shown that to generate k Taylor coefficients, we need at most $O(k^2)$ times as much computational work as we require to evaluated $f(y)$ [2]. Note that this method is still far more efficient than the standard symbolic generator of Taylor coefficients.

Since the VNODE method uses Taylor series for validated solution, the accuracy depends on the order of Taylor expansion. When a higher order is chosen, the acceptable step size can be larger, the total number of steps is reduced, however, each step requires more computation for the higher order Taylor expansion. We note that it needs the columns $T_{\text{in}}[0:n-1][0:j]$ to generate the column $T_{\text{out}}[0:n-1][j]$, then

$T_{\text{in}}[0:n-1][j+1]$. Thus, T_{in} and T_{out} are generated column by column. We have identified that this Taylor coefficient generator, the construction of the Jacobian of Taylor coefficients in particular, is the computation bottleneck of the VNODE method. It takes almost 60% of the total computational time.

Although the columns $T_{\text{out}}[0:n-1][j]$ and $T_{\text{in}}[0:n-1][j+1]$ are dependent of the previous columns $T_{\text{in}}[0:n-1][0:j]$, the entries in the same column $T_{\text{out}}[0:n-1][j]$ are independent. Our basic idea is to parallelize the computation of a column of T_{out} and distribute the load to processes, thereby to improve the performance of the VNODE method.

Multithreading programing based on shared memory provides an efficient way of parallelizing the VNODE method. In UNIX systems, there are three types of user level threads: unbounded thread, bounded thread, and light weight process (LWP). The unbound thread is the smallest size of three types of threads. It takes little time to create or wake up a unbound thread. But it is not guaranteed that different unbound threads are assigned to different CPUs. Thus, the benefit of multiprocessors may not be obvious for using unbounded thread. For bounded threads and LWPs, the operating system will try to assign threads to different CPUs so that they can be executed in parallel. In addition, LWPs provide more system calls than bounded threads, however, LWPs are much heavier than bounded threads. In our implementation, no system calls are necessary. Therefore, bounded threads can give us a good parallel scheme and introduce less overhead than LWPs.

As described above, for each step, several Taylor coefficient matrices are generated. A typical IVP of ODE requires thousands of steps. It is unfeasible to create and destroy threads in each step since thread construction and destruction are time consuming. In our parallel implementation, we use a few threads to compute the Taylor coefficients for all steps. Figure 2 shows that the main thread creates threads outside the while loop.

Each thread is responsible for one or more rows in a Taylor coefficient matrix. Let p be the number of threads created by the main thread, then thread i ($0 \le i \le p-1$) computes the rows $i, i+p, \ldots$. In other words, the rows are assigned to threads cyclically to improve load balance. Figure 3 shows the thread program.

```
Create threads;
while not finish
  Initialize Tin[0:n-1][0];
end.
```

Figure 2: Main thread program

```
for col = 0 to k-1
  for all myRow
    Tout[myRow].eval[col];
    Tin[myRow][col+1] =
      Tout[myRow][col]/(col+1);
  end
end.
```

Figure 3: Thread program

Since a few threads compute the Taylor coefficients for all steps, we have the following three synchronization issues:

- All threads must be synchronized before they start computing a new column because the computation of a column is dependent of its previous columns;
- For each step, all threads must wait until the main thread has initialized the first column of T_{in} of (4);
- After initialization, the main thread must wait until the threads complete the computation of the Taylor coefficients. Then the main thread initializes T_{in} for the next step.

We use three barriers to achieve the synchronization. Barrier1 is used for the synchronization among threads. Barrier2 blocks all threads until the main thread finishes initializing the first column of T_{in} and barrier3 blocks

```
Create threads;
while not finish
  Initialize Tin[0:n-1][0];
  barrier2;
  barrier3;
end.
```

Figure 4: Main thread program with synchronization

```
barrier2;
for col = 0 to k-1
  for all myRow
    Tout[myRow].eval(col);
    Tin[myRow][col+1] =
      Tout[myRow][col]/(col+1);
  end;
  barrier1;
end;
barrier3;
```

Figure 5: Thread program with synchronization

the main thread until the threads complete the Taylor coefficients for a step. Figure 4 shows the main thread program with synchronization and Figure 5 shows the thread program with synchronization.

Figure 6 depicts the synchronization among all threads.

To terminate the threads, a signal is set by the main thread when the final step is reached.

As we expect, barriers for synchronization introduce overhead. We gain speedup for validated method for IVPs of ODEs of sufficiently large sizes.

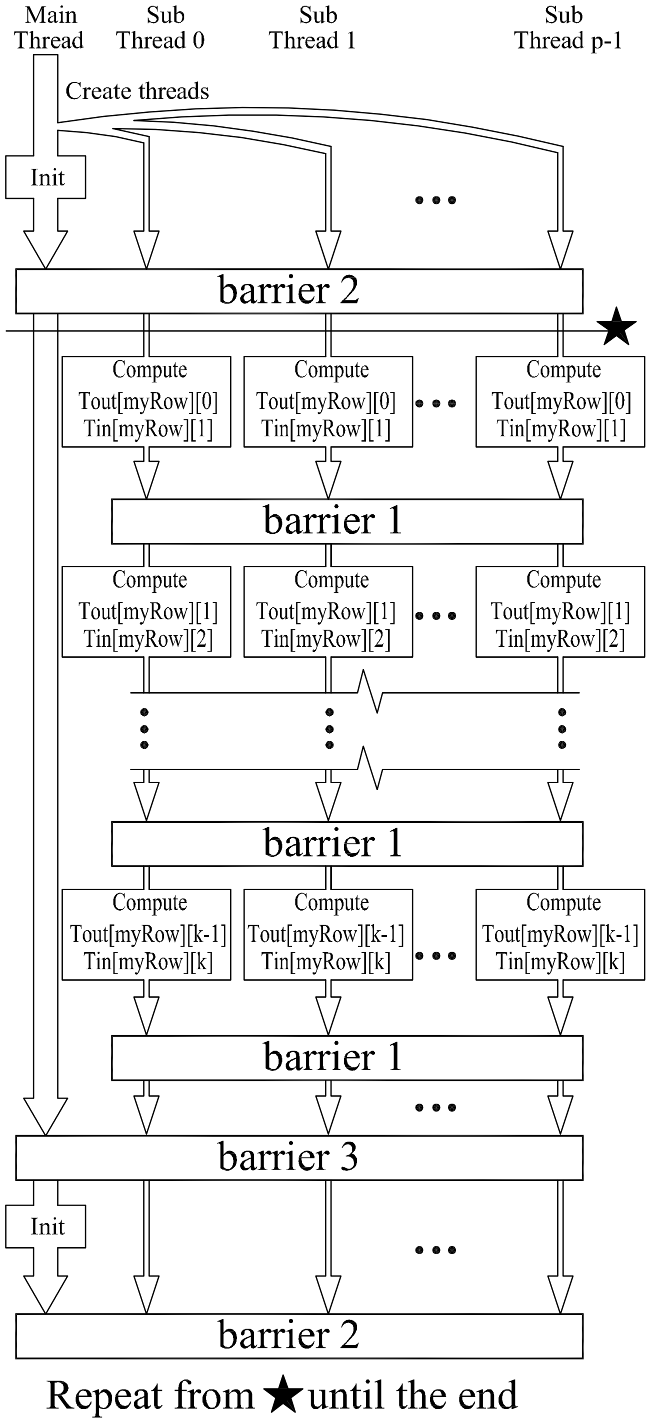

Figure 6: Sychronization among threads

References

[1] N. S. Nedialkov. *Computing Rigorous Bounds on the Solution of an Initial Value Problem for an Ordinary Differential Equation*. Ph.D. thesis, Department of Computer Science, University of Toronto, Toronto, Canada, M5S 3G4, February 1999

[2] N. S. Nedialkov, K. R. Jackson, and J. D. Pryce. An effective high-order interval method for validating existence and uniqueness of the solution of an IVP for an ODE, to appear in *Reliable Computing.*

[3] C. Bendsten and O. Stauning. TADIFF, a flexible C++ package for automatic differentiation using Taylor series. Technical Report 1997-x5-94, Department of Mathematical Modeling, Technical University of Denmark, DK-2800, Lyngby, Denmark, April 1997.

[4] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1996.

[5] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss. Validated Solutions of Initial Value Problems for Ordinary Differential Equations. *Applied Mathematics and Computation*, 105(1), 21–68, 1999.