Received 23 February 2024, accepted 5 March 2024, date of publication 11 March 2024, date of current version 15 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3375305

RESEARCH ARTICLE

A Pipeline-Based ODE Solving Framework

RUIXIA CAO[1], **SHANGJUN HOU**[2], **AND LIN MA**[3]
[1]Higher Vocational College, Dalian Neusoft University of Information, Dalian, Liaoning 116023, China
[2]Neusoft Dalian Company Ltd., Ganjingzi, Dalian, Liaoning 116085, China
[3]College of Computer Science and Technology, Jilin University, Changchun, Jilin 130012, China

Corresponding author: Ruixia Cao (cao_rx@aliyun.com)

ABSTRACT The traditional parallel solving methods of ordinary differential equations (ODE) are mainly classified into task-parallelism, data-parallelism, and instruction-level parallelism. Based on the RIDC (revisionist integral deferred correction) algorithm, a hybrid solver dispatched on both CPU and GPU is proposed, which realizes computing in a pipeline form and a remarkable parallelism is obtained both inside a single equation and among many different equations. The proposed framework can make full use of the multi-core advantage of GPU, which is conducive to load balancing within computing nodes. The efficiency and accuracy of the framework are verified in several experiments.

INDEX TERMS ODE, hybrid solution, RIDC, parallel solver.

I. INTRODUCTION

The traditional parallel solving methods often implying a sequential integrator in time, decoupling of differential equations, or the parallel evaluation of the right-hand side, primarily encounter several challenges: firstly, time-integration is inherently sequential due to causality which is inefficient; secondly, although parallel simulation hardware devices and parallel programming languages have developed rapidly, the multi-core utilization of CPU and GPU is still inefficient which highly depends on the initial value problem (IVP) and the employed parallel-in-time method; thirdly, the sequential nature of time integration does not match the existing parallel underlying support. When solving ODEs, there is an input and output dependency between the models, which determines the sequence of the solution process. Therefore, the current parallel simulation performance mainly depends on the degree of parallelism algorithms, while the algorithm software does not proceed with parallel optimization or provide an interface for parallel design, which leads to the low utilization of CPU and GPU.

Basically, there are two ways to improve the parallelism: 1. decoupling between models. After decoupling, the parallelism between systems or modules can be achieved under the conditions of convergence and stability constraints; 2. parallelism at the solver level, which uses some numerical algorithms and hardware architecture features to achieve parallelism at the data or time domain level. In this paper, we focus on the latter method to design and implement an ODE solver. The solver is based on the RIDC (revisionist integral deferred correction) algorithm, which realizes computing in a pipeline form and a remarkable parallelism. It is the first work implementing RIDC algorithm on GPU side in a pipeline form and achieving a pretty good result comparing with the existing methods, such as Ong's RIDC, Julia's RK4, and PCR in [29].

In the following sections, we will give an introduction to the related ODE solving algorithm (section II), followed by the GPU parallel solution status in section III and the software package status in section IV. Then in section V and section VI, we introduce our pipeline based ODE solving framework and the principle of RIDC Algorithm. Finally, several experiments are used to verify our framework's accuracy and efficiency in section VII and achieve some conclusions in section VIII.

II. BACKGROUND

With the limit of the hardware manufacturing process, the processor's clock frequency is close to the bottleneck and multi-core processor has become the "horizontal expansion" standard of today's computing chips. The task division and solution for multi-core processor has naturally become a hot

The associate editor coordinating the review of this manuscript and approving it for publication was Giovanni Angiulli.

issue in the current real-time computing field [1]. As a result, the time-domain parallel integration method has been given attention again. In the traditional partial differential equation-solving process, the time domain method is usually not used for parallelization, but when the parallelization in space is saturated, the time domain method provides a direction for further parallelization. However, due to the limitation of causality in time, later solutions in time are influenced by earlier solutions. Therefore, time-domain parallel algorithms are quite different from spatially-parallel algorithms, which usually require iteration in the time dimension.

Regarding the initial value problem (ODE-IVP) in parallelism in the time domain, it can be roughly divided into the following three methods proposed by Burrage, et al. [2]: the first is ''problem domain parallelism''. The ODE solving problems are divided into a series of sub-problems that can be executed in parallel and the coupling among sub-problems is realized through an iterative process. Such methods mainly include the waveform relaxation method proposed by Vandewalle, et al. [3]; the second is ''step parallelism''. The time domain is decomposed into multiple sub-time domains. Such methods include the Parareal method proposed by Lions, Gander, et al. [4], [5] by alternately applying coarse-grained sequential solvers and fine-grained parallel solvers; the third is ''method parallelism'' which integrates in each integration step (multiple functions are evaluated at the same time). This method is typically used in small-scale parallelism and is limited by the specific calculation. For example, the multi-order Runge-Kutta method is parallelized within each integration step and such solutions mainly include the methods proposed by Miranker, Enenkel, Ketcheson, et al. [6], [7], [8]. A prediction-correction framework can also be used to generate method-parallel temporal integrators, including the parallel extrapolation method proposed by Kappeller, et al. [9] and the RIDC integrator proposed by Christieb, et al. [10], [11]. Ong et al. [12] explored how time-integration methods based on deferred correction can be effective solvers on modern computer architectures and demonstrate their performance.

Specifically, the spectral deferred correction (SDC) proposed by Dutt, et al. [13] iteratively corrects the approximate solution by solving the integral formula of the error equation. This integral formula form overcomes the stability problems of traditional differential delay correction methods. SDC is a sequential method while revisionist integral deferred correction (RIDC) is a formalized variant of SDC which achieves the purpose of parallel correction by pipeline-continuous computation. Unlike SDC using Gauss-Lobatto nodes, RIDC uses evenly spaced nodes to minimize memory footprint and allows embedding of higher-order integrators. The basic idea of SDC and RIDC methods is to convert the original initial value problem (IVP) into solving the related error initial value problem and then gradually correct the numerical error in the original IVP solution; parallelization is reflected in solving the original IVP and the related error IVPs at the same time, that is, the calculation of each order's error correction is parallel. However, the majority of previous efforts have been CPU-oriented, not fully leveraging the potential of GPU. Our work explores the application of ODE solving methods on the GPU side, attempting to harness the computational power of GPU.

III. GPU PARALLEL SOLUTION STATUS

In traditional parallel ODE solvers, the algorithms are either CPU thread-oriented or incorporate GPU resources into the computational process, which has several drawbacks in the following two aspects:

1. Problem-solving areas are limited. Traditional methods of working out ODEs with homogeneous properties are only applicable to large-scale and homogeneous model instances(in the form of $y'(t) = f(t, y)$), such as group behavior evolution, distributed training simulations, etc., which can be viewed as a kind of Monte Carlo simulation method. However, many simulation models have different internal mathematical descriptions, especially for complex products in industrial areas such as automobiles, which include various sub-models of engines, hydraulics, transmissions, brakes, etc. Therefore, it is hard to build an overall spatial discrete parallel solution algorithm.

2. GPU usage is not high enough. With a whole task parallelism approach, when the number of homogeneous ODE equations is small, the portion of the model that can be run in parallel occupies only a fraction of the stream processors on the GPU, leading to a low workload. In current multi-resolution parallelism schemes, computational tasks are completely CPU dependent and GPU sits idle, which is unable to provide reasonable computational power. All these methods result in an unbalanced load on the computing resources within the computation nodes. For implicit solutions of particularly complex ODEs, the internal computation process involves fixed-point iterations. If the computation resources of the GPU cannot be effectively utilized, the computation of each global time step will consume a lot of CPU time, which slows down the simulation and reduces simulation efficiency.

Generally, there are two kinds of possible GPU-based parallel ODE solving techniques:

One direction is parallelism-across-the-task, such as, dispatching multiple instances of the same ODE equation to streaming multiprocessors on different GPUs. Except for different initial values, all these multiprocessors share the same computing logic and other parameter configurations. This parallel process relies on the SIMD feature of the GPU and requires all operations to be fully isomorphic to execute in parallel.

The technique mentioned above can be further divided into two specific implementations when solving equations in the form of $y'(t) = f(t, y)$: the first one is framework as Fig.1 [14] shows, solving each ODE through a GPU thread (WorkItem), and lots of ODEs are organized through warps(WorkGroups); the other one is exhibited in Fig.2 [15] rearranging the ODEs by the same equations so that each thread has the same

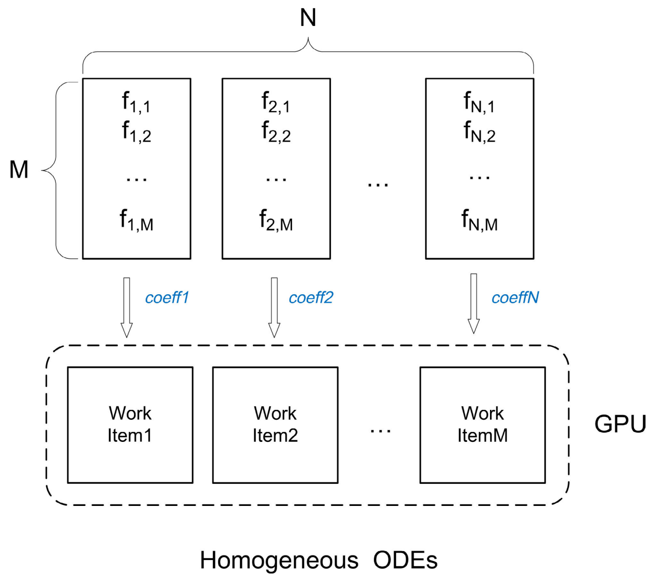

FIGURE 1. Each WorkItem corresponds to ODEs.

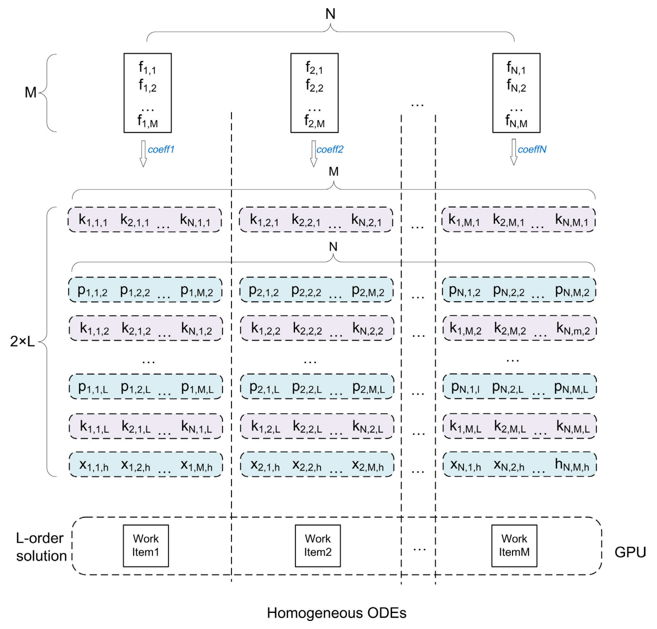

FIGURE 2. Each WorkItem corresponds to an equation.

instructions, including M sets of different ODE equations and each set contains N homogeneous equations. There are N groups of parameters and each group contains M different parameters. In the actual calculation process, N can be allocated to different work groups for execution according to the size of N. The second implementation leads to a more fine-grained task division.

Another direction is the data-parallelism, which belongs to the approach of time-domain parallel solution of IVP [16], [17]. This approach uses the iterative multi-resolution idea by estimating the initial low-resolution values in a serialized manner at first, and then performing the high-resolution corrections in parallel. This process is repeated until a reasonable accuracy is reached. Until now, this kind of parallelism is still limited to a single system of ODE equations and is suitable mainly for small-scale computations. Due to the combination of serial and parallel processing steps, it is mainly implemented on CPUs in a multi-threaded manner, no GPU version has been proposed. Based on this situation, a method that can make good use of CPU and GPU resources is needed.

IV. SOFTWARE PACKAGE STATUS

When developing quality promising software, it is essential to make adaptations to both algorithm and program code. However, in order to ensure versatility and excessive pursuit of flexibility, current software libraries often separate the relationship between the two that should be integrated, which makes it difficult to integrate new parallel algorithms. Therefore, parallel ODE solving tools still can be improved in the following technique aspects.

A. STORAGE MANAGEMENT AND DATA ORGANIZATION

Currently, the most widely adopted open-source C/C++ software package is SUNDIALS [18], [19]. Other commonly used libraries include Boost Odeint [20], GNU scientific library [21], DifferentialEquations.jl [22] and so on. Odeint is a collection of advanced numerical algorithms to solve initial-value problems of ordinary differential equations. It is written in C++ using modern programming techniques to provide high generality at optimal performance. Its main advantage over other ODE libraries is a strict separation of the numerical algorithms from the underlying arithmetical computations. This creates the possibility to use Odeint on GPUs or HPC clusters, or to combine Odeint with custom data types as well as other numerical libraries [23]. GNU provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite. Besides, it does not restrict scientific cooperation. DifferentialEquations.jl is a package for solving differential equations in Julia and offers a unified user interface to solve and analyze various forms of differential equations while not sacrificing features or performance. Among these software packages, only SUNDIALS provides the basic memory management interface [24], but the specific memory management is left to the user or third-party software. To fully leverage the parallel hardware, data needs to be closely arranged and organized in the form of vectors, which can make the most of the SIMD capability on the GPU side and the multi-threading capability on the CPU side. The CPU cache can also be used more effectively. There are two specific parallel application methods for these solvers: firstly, for large-scale systems, the solution of the equation system for dense matrices or the iterative solution for sparse matrices is dispatched to the GPU side. This kind of application generally adopts the form of directly integrating existing matrix calculation toolkits, such as cuSOLVER, ArrayFire, etc. It does not require much modification to the existing solution software and is easy to implement. Secondly, for homogeneous small model clusters (such as multi-dynamic systems distributed on spatial grid nodes), it is necessary to recombine the state variables scattered on the storage into a tightly arranged form to adapt to the memory access mode on the GPU side, which requires enhancing the memory management capabilities of existing solvers, including the ability to move data blocks as needed and eliminate the gap between the storage locations of similar state variables, that is, the power to integrate dynamic memory distribution. This integration is critical to the performance of GPU, and different implementations can lead to a performance gap more than 100 times in ODE solving [25]. The known solution tools have not achieved active memory management capabilities and required users to statically plan during initialization. Once the equation system is changed, the memory layout must be re-modified, which limits the multi-task parallel solution application in the pipeline mode. Our proposed memory management has several advantages: firstly, when faced with multi-instance ODE problems such as equations $y'(t) = f(t, y)$ in the form of the same function f with different parameters, because of ODEs' same solution logic, they have the similar state variables and Jacobi matrix. In order to make adjacent threads have access to adjacent state variables and Jacobi matrix, memory management is required; secondly, in dynamic pipeline mode, right-hand side (RHS) expression is solved on CPU side and Jacobi matrix is solved on GPU side. GPU utilization will be higher in the case of multi-threading and multi-instance since we put the same dimension equations together; thirdly, CUDA's unified memory's function is limited on Windows comparing with Linux. It's necessary to conduct memory management in our work.

B. JUST-IN-TIME COMPILATION

Just-in-time compilation technique is widely used in scripting languages or bytecode languages to improve runtime efficiency, such as python, Java, and so on. Currently, just-in-time compilation technique is mature enough and its performance is comparable to that of the native code compiled in advance. This gives the opportunity to improve the performance of ODE parallel solution for the following three reasons: firstly, the basic description form of a differential equation is a mathematical expression, and unary or binary arithmetic operations constitute its main function. Compared with C/C++ language, it does not require the optimizations of complex control flow and procedure calls, and can be directly translated into machine code. The function can be dynamically created in memory, which is convenient to call at any time; secondly, the SIMD instruction optimization for the CPU needs to be done manually at the bottom of the program. However, due to the simple arithmetic operation characteristics of mathematical expressions, this instruction optimization can be done automatically through just-in-time compilation, and the efficiency of dynamically generating and executing code is even higher than that generated by local compilation; thirdly, when faced with a large homogeneous ODE cluster, just-in-time compilation can generate code in OpenMP, CUDA or OpenCL form as needed, and adaptively allocate tasks to the CPU or GPU according to the size of the vector. Based on this, we proposed a high performance text based ODE solver with Just-In-time compiling and coupled CPU-GPU computing properties in [26].

C. DYNAMIC BINDING

A direct result of just-in-time compilation is the ease of implementing dynamic binding, which is very important for reducing the implementation complexity of some parallel algorithms. When using mainstream C/C++ solving tools, it is necessary to convert the RHS expressions and Jacobi expressions (such as using implicit arithmetic) into C language. Then, they are called by the solver in the form of function overloading or function pointer. This method belongs to static binding, that is, each time the solution object is changed, then these functions must be reimplemented and recompiled. Although this method brings convenience to programmers and it can flexibly combine solving algorithms by various means such as interface or inheritance according to specific problems, this flexibility is only reflected in the design phase, not in the runtime phase, and even seriously violates the software design concept of separating data and business logic at runtime, which brings difficulties to the dynamic adaptation of parallel algorithms. For example, the IMEX algorithm needs to switch between explicit/implicit integration algorithms according to the current rigidity characteristics of the system, and these two integration algorithms must be specified at the coding stage and bound together with the program's running logic. If the simulation has been run for a long time, it is found that the accuracy of the existing integration algorithm is deficient and needs to be switched to higher precision, or the computing power of the current computing node is insufficient and needs to be switched to a low-precision algorithm. The only solution is to stop the current simulation and restart after modifying the code. This dynamic binding is different from Python, mainly reflected in the expression evaluation function pointer can be combined with different numerical algorithms as needed. If combined with the task graph [27], more flexible load balancing can also be achieved.

According to the shortcomings mentioned in parts 2 and 3, and the open-source package, such as SUNDIALS, exhibits a lack of attention to parallel optimization, we propose a framework that combines parallelism-across-the-task and parallelism-across-the-method approach. This framework has the advantages of remarkable scalability, flexibility, load balancing, and run-time configurability, making it particularly suitable for simulation environments containing hundreds or thousands of large ODEs.

V. PIPELINE SOLVING FRAMEWORK

We design and implement a pipeline-based simulation model solving task assignment framework for CPU and GPU target

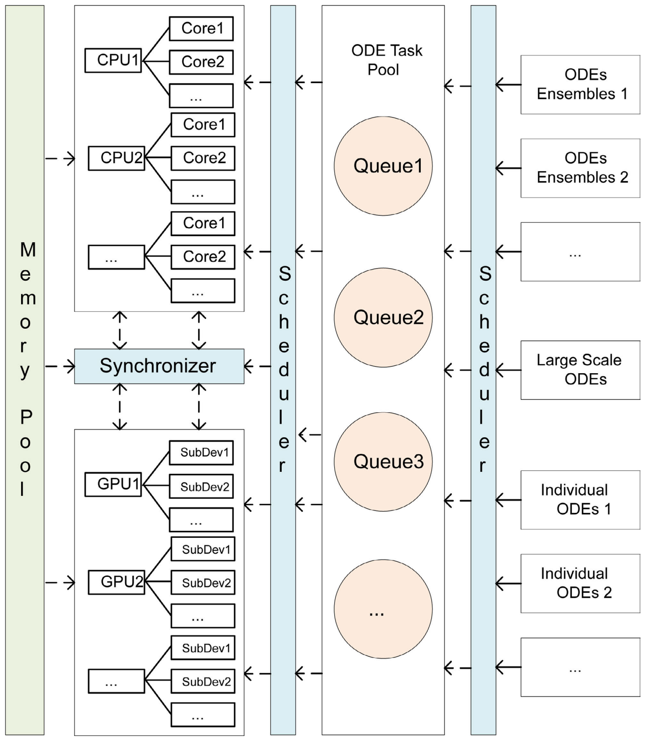

FIGURE 3. **Pipeline solving framework.**

machines to solve ordinary differential equations in parallel. This parallelism is reflected in two aspects: firstly, the RIDC algorithm is used to perform the internal parallel solution of a single equation system in the form of multiple threads; secondly, the solution process between the equations is also parallel which is similar to the traditional isomorphic multi-instance equation. Different from the GPU solution in the group form, this framework also supports the parallel solution of the heterogeneous form of ODEs which can harness the computation power of the GPU. This framework also has the ability to dynamically allocate computing resource targets. For a single ODE system, the solution forms include three modes: CPU single thread, CPU multi-threading, or GPU multi-threading. When CPU is used to solve the problem, the corresponding relationship between the number of threads and the solution accuracy can be either one-to-one or one-to-many, that is, the solution task within a thread can correspond to multiple rounds of correction within the RIDC algorithm; when using GPU to solve the equations, in order to maximize the parallel ability inside the warps, a one-to-one relationship is adopted between the solving accuracy and the thread. In addition, the internal implementation technique of this framework adopts the iterative mode in the form of prediction-correction which can better support the parallelism of many iterative methods in the calculation process, such as RIDC, Parareal, etc. Especially for those solving algorithms that require low-precision succession and high-precision parallelism, the CPU and GPU hybrid operation mode has the advantage of load balancing and can also obtain a better speedup ratio.

The core idea of this framework is to dynamically generate ODE solution tasks and schedule them to the corresponding hardware according to the solution settings. As shown in Fig.3, the framework consists of four main components: task generator, scheduler, synchronizer and memory pool. When a new ODE solution requirement arises, the task generator builds the data structure and just-in-time compilation (currently only for GPU code) routines according to the solution parameters and puts the task into the task queue in the ODE task pool. The scheduler dispatches tasks in a queue to the CPU or GPU or both of them, according to the different queue settings. For solving tasks that require both CPU and GPU, the synchronizer is responsible for data exchange between CPU and GPU. At the same time, the GPU is further divided into several sub-devices (corresponding to stream in CUDA). Each sub-device is only responsible for the solution of one ODE system or the parallel correction calculation process within one ODE system. On the CPU side, the solution task of any ODE system can be assigned to one or more threads (cores), depending on the specific solution configuration with greater flexibility. The memory pool contains two allocation methods: for individual ODEs, allocate a contiguous piece of memory for its state variables and Jacobi matrices, and support memory alignment to better adapt to hardware data access patterns; for multi-instance ODEs, contiguous memory is allocated for all instance state variables, and another contiguous memory is allocated for these Jacobi matrices to provide better support for data transmission and collaboration between CPU and GPU.

There are three types of tasks supported by this framework: the first is small-scale single ODE system. Each equation system corresponds to different models, that is, the solving tasks are different; the second is an isomorphic type of ODE system cluster which means multiple different instances of the same model. This type can harness the parallel advantages of GPU; the third type is a large and complex ODE system (tens of thousands of state variables), and the solution in this situation requires the use of multiple stream processors of the GPU.

VI. RIDC ALGORITHM

The basic principles of RIDC are as follows:

A. ERROR INITIAL VALUE

The initial value problem of ordinary differential equations (ODE-IVP) is generally expressed by the following definition:

$$\begin{cases} y'(t) = f(t, y), & t \in [0, T] \\ y(0) = y_0 \end{cases} \tag{1}$$

where $y(t) \in \mathbb{C}^N, f : \left[\mathbb{R} \times \mathbb{C}^N\right] \to \mathbb{C}^N$.

Denote the (unknown) exact solution of IVP(1) as $u(0) = y(0)$ and the approximate solution as $u(t)$. The error of the approximate solution is $e(t) = y(t) - u(t)$. Residuals (sometimes called defects) are defined as $r(t) = u'(t) - f(t, u)$, then the time derivative of the error satisfies

$$e'(t) = y'(t) - u'(t) = f(t, u + e) - f(t, u) - r(t) \quad (2)$$

According to $e(0) = u(0) - y(0) = 0$, the above equation represents the IVP of the correlation error. For the sake of stability, the error IVP is written in the integral form [6],

$$\left(e + \int_0^t r(\tau)\, d\tau\right)' = f(t, u + e) - f(t, u) \quad (3)$$

Noting that if the error equation (3) is solved numerically, the corrected approximation $u + e$ is still an approximation, we employ a more general notation that will allow us to iteratively correct the solution until the desired accuracy is achieved. Denoting the initial approximation as $u^{[0]}$, the $p-th$ approximation is expressed as $u^{[p]}$ and its error is $e^{[p]}$. Then the error equation can be rewritten as

$$\left(e^{[p]} + \int_0^t r^{[p]}(\tau)\, d\tau\right)' = f\left(t, u^{[p]} + e^{[p]}\right) - f\left(t, u^{[p]}\right) \quad (4)$$

Among which, $r^{[p]} = u^{[p]}(t)' - f\left(t, u^{[p]}\right)$.

B. DISCRETIZATION

Using algebraic methods, equation (4) can be explicitly discretized in the first order, and the solution is in the form of

$$u_{n+1}^{[p+1]} = u_n^{[p+1]} + \Delta t f\left(t_n, u_n^{[p+1]}\right) - \Delta t f\left(t_n, u_n^{[p]}\right) + \int_{t_n}^{t_{n+1}} f\left(\tau, u^{[p]}\right) d\tau \quad (5)$$

Similarly, the first-order implicit discretization of (4) is as following:

$$u_{n+1}^{[p+1]} = u_n^{[p+1]} + \Delta t f\left(t_{n+1}, u_{n+1}^{[p+1]}\right) - \Delta t f\left(t_{n+1}, u_{n+1}^{[p]}\right) + \int_{t_n}^{t_{n+1}} f\left(\tau, u^{[p]}\right) d\tau \quad (6)$$

In the semi-discretized forms (5) and (6), a sufficiently accurate orthogonal matrix is required to approximate the current integral [6]. If a first-order predictor is used to compute an approximate solution to (1) and a first-order corrector such as (5) and (6) is used, the integral can be approximated with the following formula:

$$\int_{t_n}^{t_{n+1}} f\left(\tau, u^{[p]}\right) d\tau \approx \begin{cases} \sum_{\upsilon=0}^{p+1} \alpha_{p\upsilon} f\left(t_{n+1-\upsilon}, u_{n+1-\upsilon}^{[p]}\right), & n \ge p \\ \sum_{\upsilon=0}^{p+1} \alpha_{p\upsilon} f\left(t_\upsilon, u_\upsilon^{[p]}\right), & n < p \end{cases} \quad (7)$$

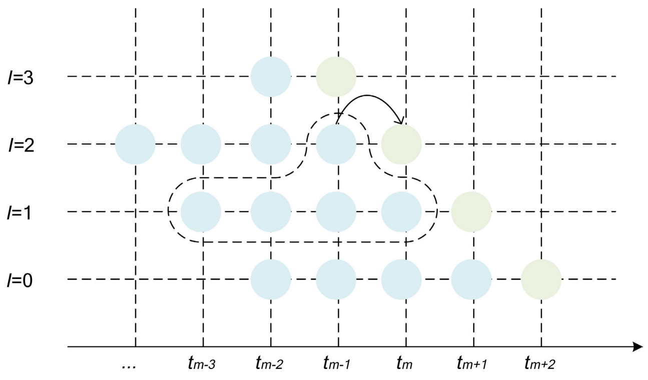

FIGURE 4. Node dependencies of each other.

where, $\alpha_{p\upsilon}$ are the orthogonal weights.

$$\alpha_{p\upsilon} = \begin{cases} \int_{t_n}^{t_{n+1}} \prod_{i=0, i\ne\upsilon}^{p+1} \dfrac{t - t_{n+1-i}}{t_{n+1-\upsilon} - t_{n+1-i}} dt, & n \ge p \\ \int_{t_n}^{t_{n+1}} \prod_{i=0, i\ne\upsilon}^{p+1} \dfrac{t - t_i}{t_\upsilon - t_i} dt, & n < p \end{cases} \quad (8)$$

C. CONVERGENCE

Regarding the convergence of the RIDC algorithm, this paper gives a conclusion without proof and the detailed proof can be found in the literature [17].

Theorem 1: Assuming that in formula (1) $f(t, y)$ and $y(t)$ are smooth enough. RIDC algorithm under $k > M + 1$ uniformly distributed nodes embeds in the r_0 order Runge-Kutta method as a prediction step and embeds $r_i\ (1 \le i \le q)$ order Runge-Kutta algorithm as correction step respectively. Then the local truncation error is $O\left(h^{SM+1}\right)$, and h is the step size. Among which, $SM = \sum_{j=0}^{M} r_i$.

It can be seen that when all the stages of the p-order RIDC algorithm are computed using first-order Euler integrators, the final truncation error is $O(hp)$.

D. CPU ALGORITHM

In order to avoid the Runge phenomenon caused by non-equidistant node integration during high-precision solution, the solution method implemented in this paper also supports the integration of non-equidistant node forms such as Gauss-Lobatt, so the space optimization form with the minimum memory footprint is not adopted in the initialization process. For the N-order accuracy requirements, the calculation dependencies between the correction nodes of each order are shown in Fig.4.

Each node solution in the higher order correction process depends on a specific number of node values inside the lower order, and the specific number of dependencies varies according to the solution accuracy. Fig.5 shows the initialization process of the fourth-order precision solution in the form of equidistant nodes. Starting from step 12, four

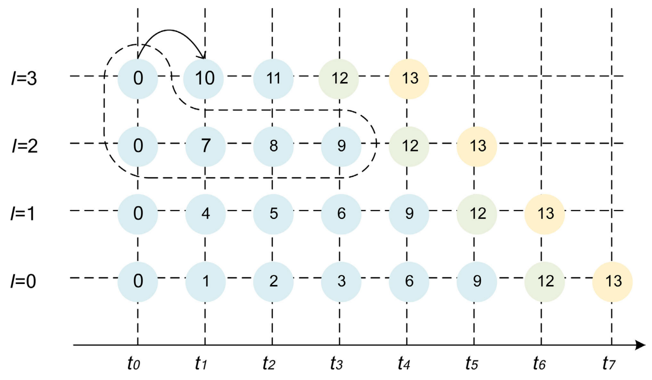

FIGURE 5. Node calculation sequence.

threads begin to execute parallel in the form of pipelines, and the final solution result of each time step is determined by the correction thread corresponding to the output of $I = 3$.

From the basic calculation process in Fig.5, it can be known that if there are k solving nodes and the solution accuracy is p order, p cores are used to solve in parallel, then the ratio of the calculation between the RIDC algorithm and the first-order algorithm is $r = 1 + M^2/k$. Where $M = p - 1$ is the number of iterative corrections. From this, it can be calculated that the theoretical solution time ratio between the fourth-order RIDC embedded Euler and the first-order Euler is $1 + 9/k$. As k increases, RIDC algorithm's calculation time approaches the first-order Euler time. However, with the augment of k, the solution error caused by extrapolation also increases. Generally, k can be taken as the value of 20~30 times M, which not only improves the efficiency, but also ensures a certain accuracy.

The CPU parallel solution algorithm is as following:

The CPU solution algorithm supports a flexible thread allocation mode, which can be either single-thread sequential or multi-thread parallel. The maximum parallel number of threads solution is equal to the order of the solution precision. Using the maximum parallel mode can reduce the calculation time, and the wall time of single-step progress is close to other serial algorithms with the same order.

E. GPU ALGORITHM

In the GPU solution algorithm, according to the choice of explicit and implicit algorithms, the process contains multiple calls of different kernel functions. For the sake of brevity, it is not reflected in Algorithm 2. The parallel mode inside the GPU is subdivided into two cases: firstly, for the non-Newton iteration part in the explicit algorithm and the implicit algorithm, the parallel number of GPU threads is consistent with the solution precision order, and the calculation process is completed by a single CUDA stream; secondly, for the Newton iteration part in the implicit solution, since different linear equations need to be solved at the same time, it is necessary to assign the solution of each equation to a different stream. Go to the first parallel mode again, after all streams are executed.

Algorithm 1 CPU Parallel Solution

```
Require:
    Solution form (explicit or implicit), ODE function pointer, Jacobi matrix function pointer (needed for implicit solution), solution precision (order) ORDER, number of parallel threads NT, initial value of state variable X0, start and end time, step size.
Ensure:
    State variable value X corresponding to the current step.
 Initialization:
 1: Initialize a static thread pool with NT threads.
 2: Calculate the number of iterations required for the given step size and total time, and store it in a variable NUM_STEPS.
 3: for i = 1 to ORDER do
 4:   if $i == 1$ then
 5:     Predict state variables X from step 1 to step (ORDER - 1) * (ORDER - 1).
 6:   else
 7:     Correct state variables X from step 1 to step (i - 1) * (ORDER - 1).
 8:   end if
 9: end for
10: Initialize current step number STEP to 1.
11: while STEP <= NUM_STEPS do
    CPU threads solve in parallel:
12:   Assign thread 1 with prediction task (order 1).
13:   Assign each thread in the pool with some specific correction task (order > 1) in an interleaved fashion.

14:   for each thread in the pool do
15:     if $order == 1$ then
16:       Predict state variables X from current step to step current + ORDER - 1.
17:     else
18:       Correct state variables X from current step to step current + ORDER - 1.
19:     end if
20:   end for
21:   Wait for all threads to complete computation.
22:   for each thread in the pool do
23:     Left shift state variables X in memory by ORDER - 1 positions.
24:   end for
25:   Output state variables X of current time step.
26:   Increase current step number STEP.
27: end while
28: Perform shutdown routine, clean up thread pool and other resources.
```

F. MULTIPLE INSTANCE ALGORITHM

The design of the current solver is based on ODE system. No matter whether it is a heterogeneous equation system or multiple different instances of a homogeneous equation

Algorithm 2 GPU Parallel Solution

```
Require:
    Solution form (explicit or implicit), ODE global function
    pointer, Jacobi matrix global function pointer (needed
    for implicit solution), solution precision (order) ORDER,
    initial value of state variable X0, start and end time, step
    size.
Ensure:
    State variable value X corresponding to the current step.
 Initialization:
 1: dispatch one CPU thread in the dynamic thread pool as a
    worker
 2: initialize host and device memory needed(include global
    and local device memory)
 3: copy state variables X0 to device memory
 4: create GPU streams(one stream for explicit solver,
    or ORDER streams for implicit solver)
 Solve in parallel:
 5: for each GPU thread in the warps(warps number =
    ORDER) do
 6:    if $thread\ id == 1$ then
 7:       predict initial value of state variables X by explicit
          solver
 8:       if implicit solver then
 9:          call NewTon-Raphson method to get a refined
             value in an iterative manner
10:       end if
11:       loop from current step to current + ORDER -1 step
12:    else
13:       predict initial value of state variables X by explicit
          solver and quadrature marix
14:       if implicit solver then
15:          call NewTon-Raphson method to get a refined
             value in an iterative manner
16:       end if
17:       loop from current step to current + ORDER -1 step
18:    end if
19: end for
20: __syncthreads()
21: for each thread in the warps do
22:    left shift state variables X in the memory with ORDER
       -1 position
23: end for
24: __syncthreads()
25: output state variables X of current time step
```

system, it always corresponds to one solver instance. Therefore, different from the traditional GPU solution method, this solver can support CPU and GPU parallel solution or hybrid solution of large-scale heterogeneous ODEs, and the specific number of parallel solutions is only limited by the hardware resource tolerance of CPU and GPU. The following table shows the GPU parallel scheduling pseudo code for multiple heterogeneous equation groups. Each thread in the

Algorithm 3 GPU Parallel Solving of Heterogeneous ODE or Multi-Instance ODE

```
1: initialize static thread pool with NT threads
2: for each thread in the thread pool do
3:    assign each thread a GPU ODE solver
4:    initialize GPU ODE solver
5: end for
6: parallel executing GPU solver
7: callback on each global step
8: readpool.wait()
```

CPU thread pool is responsible for the GPU solution process scheduling of a specific ODE equation group, and calls back at the global time step to complete the output of state variables or other forms of data interaction between models.

VII. EXPERIMENT

In order to verify the accuracy and efficiency of our proposed pipeline framework, we design several experiments to test the model performance. The experiments are conducted on Visual Studio 2019, Intel(R) Core(TM) i9-10900K CPU and NVIDIA GeForce RTX 3080 GPU. First, we use the following equations (9) to verify the framework's precision:

$$\begin{cases} \dfrac{dx}{dt} = rx \\ x(0) = x_0 \end{cases} \tag{9}$$

We can obtain the ordinary differential equation of the above formula by mathematical method as $x(t) = x_0e^{rt}$. To simplify the calculation, assuming that $r = 1, x_0 = 1$, then the equation can be simplified as $x(t) = e^t$. The experiment is based on the fourth-order Euler algorithm and fourth-order Runge-Kutta method, and we compare the results with the mathematical solution. The start time and end time is set to 0.0 and 10.0, the step size is 0.1. The results in our experiments and the average error is 0.00% which indicates the accuracy of our program.

In the second experiment, we use 32 groups of two equations with 1000, 10000 and 100000 sampling point, comparing explicit Fourth-order Euler and Fourth-order Runger-Kutta on the CPU side, Julia's RK4 and CPU version RIDC in [28] with explicit Euler algorithm on the GPU side, the equations are as eq.10 shows, then set the equations number to 10 and conduct the same experiment. The results are shown in TABLE 1 and 2 respectively.

$$\begin{aligned} &y'[0] \\ &= sin(\pi * X[0]) * cos(sqrt(10 * X[1])) * sin(0.3 * \pi * \\ &\quad X[1])/cos(-5 * X[0]) * sin(\pi * X[1]) * cos(sqrt(-0.2 * \\ &\quad X[1])) * sin(0.1 * \pi * X[0])/cos(0.3 * X[0]) \\ &y'[1] \\ &= -sin(0.1 * \pi * X[1])/cos(-5 * X[0]) * sin(\pi * X[0]) \\ &\quad * cos(sqrt(10 * X[1])) * sin(0.2 * \pi * X[1])/cos(-3 * \\ &\quad X[0]) * sin(0.2 * \pi * X[0]) * cos(sqrt(3 * X[1])) \end{aligned} \tag{10}$$

TABLE 1. 32 groups of two equations on CPU and GPU explicit algorithm time comparison(Time: s).

Number of samples	Ong's RIDC	Julia's RK4	CPU single thread explicit 4th-order Euler	CPU single thread 4th-order Runger-Kutta	GPU explicit 4th-order Euler
1000	0.128	0.192	0.143	0.048	0.171
10000	1.12	2.08	1.753	0.496	1.620
100000	11.2	26.112	17.236	4.933	15.758

TABLE 2. 32 groups of ten equations on CPU and GPU explicit algorithm time comparison(Time: s).

Number of samples	Ong's RIDC	Julia's RK4	CPU single thread explicit 4th-order Euler	CPU single thread 4th-order Runger-Kutta	GPU explicit 4th-order Euler
1000	0.448	0.704	0.283	0.123	0.34
10000	4.512	7.424	2.72	1.25	3.26
100000	44.384	82.592	28.36	12.41	29.85

TABLE 3. 32 groups of two equations on CPU and GPU implicit algorithm time comparison(Time: s).

Number of samples	Ong's RIDC	CPU implicit 4th-order Euler	GPU implicit 4th-order Euler
100	0.031	0.067	0.027
1000	0.294	0.651	0.248

TABLE 4. 32 groups of ten equations on CPU and GPU implicit algorithm time comparison(Time: s).

Number of samples	Ong's RIDC	CPU implicit 4th-order Euler	GPU implicit 4th-order Euler
100	0.058	0.132	0.043
1000	0.583	1.342	0.451

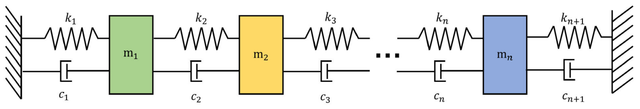

FIGURE 6. Mass spring-damper system.

From TABLE 1 and TABLE 2, it can be seen that the two equations time-to-solution rank is Julia's RK4 > our GPU version > Ong's RIDC; when comes to 10 equations, the time-to-solution rank is Julia's RK4 > Ong's RIDC > our GPU version, which indicates our proposed ODE solving framework is efficient. Therefore, in the case of multi-instance, the GPU version has advantages, which is especially suitable for solving calculations in the aspect of population evolution, such as emergence problems, molecular chemical reactions and so on.

Then in the third experiment, since RK4 is not a implicit algorithm, when conducting implicit algorithm test, it is excluded, and Ong's RIDC is 4th order. Changing explicit Euler algorithm to implicit Euler algorithm in the program and do the same test, results are shown in TABLE 3 and 4.

We also simulated a mass spring-damper system(a kind of multi-rigid-body system) which is shown in Fig.6. Use $k_i, i \in \{1, N+1\}$ as the spring constant, complying with $500 \sim 800$ uniform distribution, $c_i, i \in \{1, N+1\}$ denotes the damping coefficient from $200 \sim 400$ uniform distribution and $m_i, i \in \{1, N+1\}$ represents the mass ranging from $50 \sim 100$. In our program, N represents the number of

TABLE 5. Different thread applied in mass spring-damper system.

Method	N	Thread	Time(s)
our method	20	32	32.841
our method	20	64	16.691
our method	20	96	10.962
our method	20	128	8.919
PCT	20	200(iterations)	1096.9

the rigid bodies. The experiment is compared with [29] in TABLE 5 and the equations(equals to the stream size) are as follows. The global memory is set to $ORDER * ORDER * ORDER * N$.We need to solve both displacement d_i and velocity v_i of each element, and impose zero initial conditions for both the displacements and the velocities.

$$\begin{aligned} \dot{d}_i &= v_i \\ \dot{v}_i &= \frac{K_i}{M_i}(d_i - d_{i-1}) - \frac{K_{i+1}}{M_{i+1}}(d_{i+1} - d_i) \\ &\quad + \frac{C_i}{M_i}(v_i - v_{i-1}) - \frac{C_{i+1}}{M_{i+1}}(v_{i+1} - v_i)) + f_i \end{aligned} \tag{11}$$

where,

$$f_i = \begin{cases} f(t), i = 1 \\ 0, i > 1 \end{cases} \tag{12}$$

and $f(t) = \sin(\frac{2\pi}{T})$.

Through the above experiments, it can be found that the framework designed in this paper can significantly improve the solution efficiency of ordinary differential equations. For explicit Euler algorithm, GPU solution time is close to CPU single thread solution; for the implicit Euler algorithm, the GPU solution time is less than the CPU single thread solution. With the increase of the number of equations and the order of the solution, the effect of the framework is improved more significantly. In addition, the number of CPU cores is generally only 16, while the number of GPU cores is over ten thousand, which is hundred times of CPU cores. Therefore, it is believed that the pipeline-based simulation model solving task allocation framework for CPU and GPU target machines can greatly improve the solving efficiency of ordinary differential equations.

VIII. CONCLUSION

In this paper, a pipeline solving framework for CPU and GPU target machine is designed and implemented, which can effectively reduce the solving time and improve the solving efficiency. Especially when faced with complex simulation systems, it can take full use of the multi-core advantage of GPU. Through several experiments, the efficiency and accuracy of the framework are verified. In the future, we will further improve the computing efficiency and concurrency capabilities of the framework by computing the pre-allocation of streams and conducting device memory data exchange methods among equation groups.

REFERENCES

[1] M. S. Nobile, P. Cazzaniga, D. Besozzi, and G. Mauri, "GinSODA: Massive parallel integration of stiff ODE systems on GPUs," *J. Supercomput.*, vol. 75, no. 12, pp. 7844–7856, Dec. 2019.

[2] K. Burrage, "Parallel methods for ODEs," *Adv. Comput. Math.*, vol. 7, no. 1, pp. 1–31, 1997.

[3] S. Vandewalle and D. Roose D, "The parallel waveform relaxation multigrid method," in *Proc. SIAM Conf. Parallel Process. Sci. Comput.* Philadelphia, PA, USA: SIAM, 1987, pp. 152–156.

[4] J. L. Lions, Y. Maday, and G. A. Turinici, "A 'parareal' in time discretization of PDE's," *Comptes Rendus Acadmie Sci.-I-Math.*, vol. 332, no. 7, pp. 661–668, 4442.

[5] M. J. Gander and S. Vandewalle, "On the superlinear and linear convergence of the parareal algorithm," in *Domain Decomposition Methods in Science and Engineering XVI*. Berlin, Germany: Springer, 2007, pp. 291–298.

[6] W. L. Miranker and W. Liniger, "Parallel methods for the numerical integration of ordinary differential equations," *Math. Comput.*, vol. 21, no. 99, p. 303, Jul. 1967.

[7] R. F. Enenkel, *DIMSEMS: Diagonally Implicit SingleEigenvalue Methods for the Numerical Solution of Stiff Ordinary Differential Equations on Parallel Computers*. Toronto, ON, Canada: Univ. Toronto, 1997.

[8] D. Ketcheson and U. B. Waheed, "A comparison of high-order explicit Runge–Kutta, extrapolation, and deferred correction methods in serial and parallel," *Commun. Appl. Math. Comput. Sci.*, vol. 9, no. 2, pp. 175–200, Jun. 2014.

[9] M. Kappeller, M. Kiehl, M. Perzl, and M. Lenke, "Optimized extrapolation methods for parallel solution of IVPs on different computer architectures," *Appl. Math. Comput.*, vol. 77, nos. 2–3, pp. 301–315, Jul. 1996.

[10] A. Christlieb, B. Ong, and J.-M. Qiu, "Integral deferred correction methods constructed with high order Runge–Kutta integrators," *Math. Comput.*, vol. 79, no. 270, pp. 761–783, Sep. 2009.

[11] A. Christlieb and B. Ong, "Implicit parallel time integrators," *J. Sci. Comput.*, vol. 49, no. 2, pp. 167–179, Nov. 2011.

[12] B. W. Ong and R. J. Spiteri, "Deferred correction methods for ordinary differential equations," *J. Sci. Comput.*, vol. 83, no. 3, pp. 1–29, Jun. 2020.

[13] A. Dutt, L. Greengard, and V. Rokhlin, "Spectral deferred correction methods for ordinary differential equations," *BIT Numer. Math.*, vol. 40, no. 2, pp. 241–266, 3835.

[14] K. E. Niemeyer and C. J. Sung, "GPU-based parallel integration of large numbers of independent ODE systems," in *Numerical Computations With GPUs*. Cham, Switzerland: Springer, 2014, pp. 159–182.

[15] K. Ahnert, D. Demidov, and M. Mulansky, "Solving ordinary differential equations on GPUs," in *Numerical Computations With GPUs*. Cham, Switzerland: Springer, 2014, pp. 125–157.

[16] M. J. Gander, "50 years of time parallel time integration," in *Multiple Shooting and Time Domain Decomposition Methods: MuS-TDD, Heidelberg, May 6–8, 2013*. Cham, Switzerland: Springer, 2015, pp. 69–113.

[17] A. J. Christlieb, C. B. Macdonald, and B. W. Ong, "Parallel high-order integrators," *SIAM J. Sci. Comput.*, vol. 32, no. 2, pp. 818–835, Jan. 2010.

[18] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 363–396, 2005.

[19] Z. Ding, H. Shen, and J. Cao, "Parallel computation of continuous Petri nets based on hypergraph partitioning," *J. Supercomput.*, vol. 62, no. 1, pp. 345–377, Oct. 2012.

[20] K. Ahnert and M. Mulansky, "Odeintsolving ordinary differential equations in C++," in *Proc. AIP Conf.*, 2011, vol. 1389, no. 1, pp. 1586–1589.

[21] M. Galassi, J. Davies, and J. Theiler, *GNU Scientific Library*. Godalming, U.K.: Network Theory Limited, 2002.

[22] C. Rackauckas and Q. Nie, "DifferentialEquations.Jl—A performant and feature-rich ecosystem for solving differential equations in Julia," *J. Open Res. Softw.*, vol. 5, no. 1, p. 15, May 2017.

[23] M. Mulansky and K. Ahnert, "Odeint library," *Scholarpedia*, vol. 9, no. 12, p. 32342, 2014.

[24] C. J. Balos, D. J. Gardner, C. S. Woodward, and D. R. Reynolds, "Enabling GPU accelerated computing in the SUNDIALS time integration library," *Parallel Comput.*, vol. 108, Dec. 2021, Art. no. 102836.

[25] D. Nagy, L. Plavecz, and F. Hegedıs, "Solving large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs: Performance comparisons of MPGOS, ODEINT and DifferentialEquations.Jl," 2020, *arXiv:2011.01740*.

[26] L. Ma, X. Zhang, and X. Lei, "A high performance text based ODE solver for flexible computing task assignment," in *Proc. 4th Int. Conf. Adv. Comput. Technol., Inf. Sci. Commun. (CTISC)*, Apr. 2022, pp. 1–5.

[27] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "TaskFlow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 6, pp. 1303–1320, Jun. 2022.

[28] B. W. Ong, R. D. Haynes, and K. Ladd, "Algorithm 965: RIDC methods: A family of parallel time integrators," *ACM Trans. Math. Softw.*, vol. 43, no. 1, pp. 1–13, Mar. 2017.

[29] M. C. Messner, T. Hu, and T. Chen, "Time-vectorized numerical integration for systems of ODEs," 2023, *arXiv:2310.08649*.

RUIXIA CAO received the M.S. degree from Dalian Maritime University, in 2009. She is currently a professional Teacher with Dalian Neusoft University of Information. Her research interests include artificial intelligence and big data, including theory, models, and algorithms.

SHANGJUN HOU received the B.S. degree from Xi'an Communication University, in 2006. He is currently a Consulting Expert of the Neusoft Education Technology Group. His research interest includes big data, including data analysis and mining, and data visualization.

LIN MA received the B.E. degree in electronic information science and technology and the master's degree from the College of Computer Science and Technology, Jilin University, in 2017 and 2023, respectively, where he is currently pursuing the Ph.D. degree.