计算机科学
COMPUTER SCIENCE

基于线程池任务调度的局部同步FMI联合仿真方法

薛朝阳, 钱晓超, 刘飞

引用本文

薛朝阳, 钱晓超, 刘飞. 基于线程池任务调度的局部同步FMI联合仿真方法[J]. 计算机科学, 2026, 53(2): 107-116.

XUE Zhaoyang, QIAN Xiaochao, LIU Fei. Local Synchronous FMI Co-simulation Method Based on Thread Pool Task Scheduling [J]. Computer Science, 2026, 53(2): 107-116.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.250200061

基于线程池任务调度的局部同步 FMI 联合仿真方法

薛朝阳[1]　钱晓超[2]　刘　飞[1]
1 华南理工大学软件学院　广州 515000
2 上海机电工程研究所　上海 201109
(202221045693@mail.scut.edu.cn)

摘　要　并行仿真是提高仿真性能的关键技术,然而当前基于 FMI(Functional Mock-up Interface)的并行联合仿真存在诸多挑战,如 FMU(Functional Mock-up Unit)和执行线程耦合、输入输出同步、FMI API 互斥等问题。针对这些问题,提出了一种基于线程池任务调度的局部同步 FMI 联合仿真方法。首先,给出了该方法的框架,该框架由仿真方案、主算法、调度器和缓冲区组成,以提供并行仿真的模块化表示。然后,重点描述了 FMI 并行联合仿真主算法,其将单个 FMU 的单次仿真任务分为仿真执行和任务调度两阶段,由调度器调度执行,并自定义读写锁解决仿真过程中的同步问题;同时,通过将输出暂存到缓冲区,以避免 FMI API 竞争访问的问题。最后通过一个房间温差模型和一个船舶定位模型验证了所提方法的准确性,与 FMU 并行的非迭代雅可比方法进行对比,所提方法取得了显著的性能提升。

关键词:并行仿真;联合仿真;FMI;FMU;主算法;无锁优化
中图分类号　TP391.9

Local Synchronous FMI Co-simulation Method Based on Thread Pool Task Scheduling

XUE Zhaoyang[1],QIAN Xiaochao[2] and LIU Fei[1]
1 School of Software Engineering,South China University of Technology,Guangzhou 515000,China
2 Shanghai Institute of Mechanical and Electrical Engineering,Shanghai 201109,China

Abstract　Parallel simulation is a key means to improve simulation performance. However,parallel co-simulation based on FMI faces many challenges,such as coupling between FMU and threads,input/output synchronization,and mutual exclusion of FMI API. In response to these problems,this paper proposes a local synchronous FMI co-simulation method based on thread pool task scheduling. Firstly,the framework of the method is presented,consisting of a simulation scheme,master algorithm,scheduler,and buffer to provide a modular representation of parallel simulation. Then,the master algorithm of FMI parallel co-simulation is described,which divides a simulation task of a FMU into two stages:simulation execution and task scheduling. The scheduler schedules the task to execute. And customizes read-write lock is designed to solve the synchronization problem during the simulation. The output is temporarily stored in a buffer to solve the problem of FMI API contention for access. The accuracy of the proposed method is verified through a room temperature difference model and a ship positioning model. Compared with the non-iterative Jacobi method parallel to FMU,significant performance improvement is achieved.

Keywords　Parallel simulation,Co-simulation,Functional Mock-up Interface,Functional Mock-up Unit,Master algorithm,Lock-free

1　引言

联合仿真技术通过组合同构/异构仿真单元实现耦合系统的全局仿真,为多物理域系统集成和运行提供了解决方案[1]。为了屏蔽仿真单元的内部实现,允许不同的建模工具生成的仿真模型进行交互,Modelisar[2]提出了独立于建模软件的接口标准 FMI,即将仿真模型封装为功能模型单元(FMU)。FMU 可由 Simulink,OpenModelica 等通用的建模工具实现,表现为提供输入、输出、仿真等接口的黑盒模型。联合仿真工具负责配置各 FMU 变量间的依赖关系,并通过主算法(Master Algorithm,MA)对 FMU 进行编排执行、变量交换、误差估计、记录输出等。

到稿日期:2025-02-17　返修日期:2025-05-23
基金项目:国家自然科学基金(62273153);广东省基础与应用基础研究基金(2024A1515010900)
This work was supported by the National Natural Science Foundation of China(62273153) and Guangdong Basic and Applied Basic Research Foundation(2024A1515010900).
通信作者:刘飞(feiliu@scut.edu.cn)

在单机环境下的 FMI 联合仿真过程中，主算法在调度、通信和数据同步方面仍存在性能瓶颈。特别是在参与仿真的 FMU 数量较多或耦合关系复杂时，仿真效率和资源利用率均难以满足工程级应用需求。此外，当前多数主控策略以串行或静态分配为主，缺乏对 FMU 依赖关系和计算负载的动态适配机制。因此，围绕单机环境中主控算法的性能优化，研究高效调度策略、并行执行机制与资源调度算法，具有重要的工程实用价值。FMI 并行联合仿真面临的挑战主要表现在以下方面。

1）现有的 FMU 并行化方法[3-7]在仿真执行前，需要为 FMU 指定固定的线程，由该线程执行该 FMU 的相关操作。然而，在大型仿真场景中，显式为所有 FMU 指定执行线程较为繁琐，并且很难保证负载均衡。因此，针对大型仿真场景，需要一种自动化编排方法以合理调度 FMU 仿真执行。

2）FMI 并行仿真的主要挑战在于需要在多个处理器上合理编排 FMU 仿真，以保证其仿真步进和数据交换顺序的准确性。由于各 FMU 的仿真步长可能不同，当存在数据依赖的多个 FMU 步长不相等时，将引入仿真误差。因此，确定 FMU 的仿真步长和数据交换时机成为联合仿真的一个关键问题。

3）FMI API 接口存在的互斥问题。FMI 的标准接口（get，set，do_step）共享部分资源，如果两个操作同时执行，这些资源可能被破坏。因此 FMI 的标准函数不是线程安全的，即不能并行执行同一个接口。为解决该问题，图并行化技术[8-11]以 FMI 的具体操作为节点，保证同一 FMU 的接口按拓扑先后关系执行。然而该方法会导致任务粒度过细，会在任务调度上产生大量时间开销。

为应对上述并行联合仿真存在的挑战，本文重点探索主算法的并行优化和动态资源调度机制，提出了基于线程池任务调度的局部同步 FMI 联合仿真方法。该方法将 FMU 的仿真操作封装为任务，由线程池调度执行。通过步长约束、同步策略和无锁操作对仿真算法进行优化，进而提高大型仿真场景下的联合仿真性能。本文的主要贡献在于：

1）使用局部同步代替全局同步。FMU 的仿真步进和步长修正只受到前置和后置 FMU 的影响。本文根据前后 FMU 的依赖计数判定是否执行下一次仿真调度，避免了多线程场景下全局同步的开销。

2）使用线程池调度仿真任务。将 FMU 的单次仿真封装为仿真执行和调度判断两阶段。FMU 不与特定线程绑定，而是由线程池调度执行。无需指定 FMU 和执行线程的关系，避免了在大型仿真场景下用户线程数量过多导致性能开销过大的问题。

3）自定义读写锁以解决并行仿真的同步问题。为保护 FMI API 和缓冲，本文方法在仿真执行前为 FMU 加写锁，在调度判断阶段前为 FMU 加读锁，并通过原子操作优化锁同步，降低线程同步开销。

2 相关工作

FMI 并行仿真算法可大致分为有向无环图（DAG）并行化调度算法和 FMU 并行化调度算法两大类。其中，图并行化调度算法[8-11]以 FMU 相关操作为节点，以 FMU 端口的输入输出依赖关系确定边，建立仿真的有向无环图，进而使用经典 DAG 并行调度算法执行仿真。FMU 并行调度算法[3-7]为每个 FMU 仿真指定执行核心以实现并行，并在全局时间点交换 FMU 的输入输出。FMU 并行化调度算法基于经典 FMI 主算法[12-14]，所有 FMU 在仿真过程中使用全局仿真步长，步长大小的选择在精度与执行效率之间权衡。例如，DACCOSIM[5]根据前一次步进的仿真结果的精度动态调整步长。在并行性方面，FMU 并行调度算法为每个 FMU 分配单独的线程，以保持较好的局部性。然而，当参与仿真的 FMU 数量增加时，线程数量也会增加，而线程数量超过系统核心数量时，将导致仿真效率降低。

仿真单元（FMU）并行化是当前应用较为广泛的并行联合仿真方法，在多个领域的联合仿真场景中表现出良好的适用性。Wu 等[15]使用 OpenMP 为列车动力学模型和流体力学模型分别分配了单独的计算线程，实现了重载列车制动系统联合仿真的并行加速。Shen 等[16]提出离散元法与多体动力学的耦合框架，并通过模型并行化验证了该框架的可行性。Zheng 等[17]将两个开源电力网格仿真器耦合在分层大型基础设施联合仿真引擎（HELICS）上，实现了传输电网的快速并行动态仿真。仿真单元并行化在多个领域的联合仿真实践中展现出良好的应用效果。然而，联合仿真存在并行优化的空间。虽然 FMU 并行化算法能适配已有的联合仿真算法，但是需要为每个 FMU 分配单独的线程或核心，在 FMU 数量超过系统核心数时，将导致性能下降。同时，全局时间同步也无法充分发挥联合仿真并行性。而图并行化算法能够避免全局时间同步的开销，但是需要对同一个 FMU 的并行操作进行同步，导致任务粒度过细，因此该方法主要面向离线仿真场景。

近年来，并行计算的研究取得了显著进展。在任务调度优化方面，Stewart 等[18]提出了针对具有通信延迟的任务调度问题的双目标混合整数规划模型，为并行计算中的最大完工时间和能耗提供了优化方向。Li 等[19]采用 fork-join 模型扩展线程池任务管理，并通过非阻塞通信协议和任务窃取机制实现了粒子群优化问题的加速。在无锁优化方面，索引和队列等通用数据结构的无锁并行优化也取得了一系列研究成果[20-21]。自适应任务调度和无锁优化技术为复杂系统的并行联合仿真性能优化提供了新的技术路径和实现可能。

3 FMI 并行联合仿真框架

针对 FMI 并行联合仿真存在的自动编排、依赖约束、接口互斥等问题，本文提出了一种基于线程池任务调度的局部同步并行联合仿真方法，其总体框架和主要组成如图 1 所示。该方法将 FMU 的一次仿真封装为一个调度任务，推送到任务队列中，由线程池中的线程获取并执行。仿真任务包括仿真执行和调度判断两个阶段。在仿真执行阶段，通过写锁保证 FMU 接口的互斥性，并执行一次仿真步进。在调度判断阶段，根据仿真方案图中的输入输出依赖计数以及前置和后

置 FMU 的当前仿真时间，决定是否调度对应的 FMU 执行下一个仿真步长。

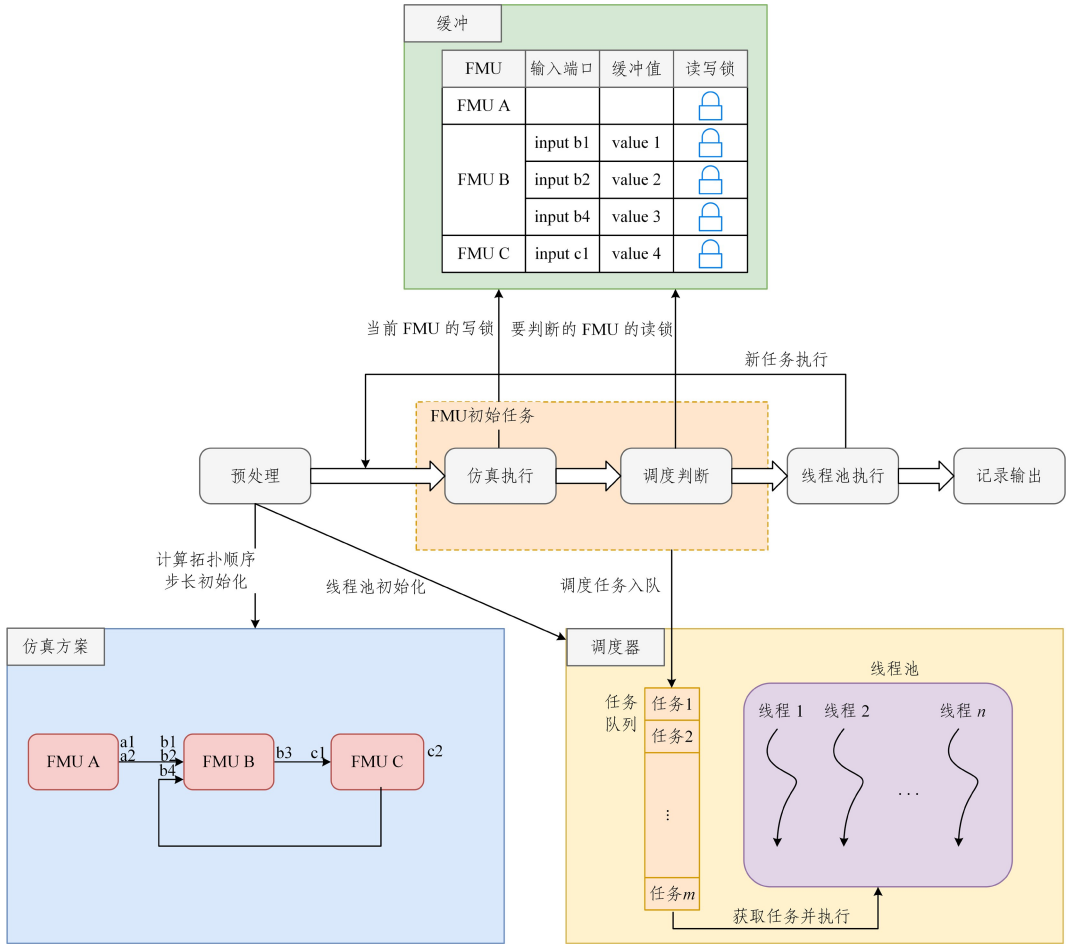

图 1 FMI 并行联合仿真框架

Fig. 1 Framework of FMI parallel co-simulation

仿真框架包含仿真方案、调度器、缓冲和主算法。仿真方案作为联合仿真的输入，描述了 FMU 及其端口依赖。调度器由线程池实现，线程获取并执行主算法推送的仿真任务。缓冲是 FMU 的输入变量缓冲，避免多线程直接访问 FMI API，并由读写锁保护缓冲区的变量以及 FMI 相关 API 调用。主算法执行仿真并记录输出。

3.1 调度器

本文方法使用任务调度策略以避免全局的时间同步。将 FMU 的一次仿真执行和调度判断封装为一个任务，由调度器自动编排和调度仿真任务执行。在仿真开始时，将每个 FMU 在仿真时间为 0 的仿真执行和调度判断任务放置到任务队列中，并启动调度器执行任务。FMU 在调度判断时，根据前置节点和后置节点的依赖计数和仿真时间，决定是否调度相关节点执行后续仿真。如果满足调度条件，则将对应 FMU 的下一次仿真调度任务放置到任务队列中等待调度。

计算线程数是调度器的重要参数，决定了仿真性能和 CPU 开销。FMU 的仿真执行受到前置和后置 FMU 的仿真时刻的约束，且受 FMU API 互斥的影响，在同一时刻同一 FMU 的仿真任务无法由多个线程进行调度。因此，算法的加速比无法超过参与仿真的 FMU 的数量，调度器的线程数量最多为仿真方案中的 FMU 的数量。此外，当线程数量增加时，多线程竞争任务队列的加锁和解锁的开销也会随之增加。综上，将线程池的线程数量作为一个可配置参数，根据系统核心数等参数和仿真方案中的 FMU 数量自定义配置。

3.2 步长约束

为了避免仿真误差导致的回滚以及引起的迭代仿真，本文方法使用了 Eguillon 等[22]提出的步长约束策略，该策略包括基于输入的约束和基于输出的约束，如图 2 所示。

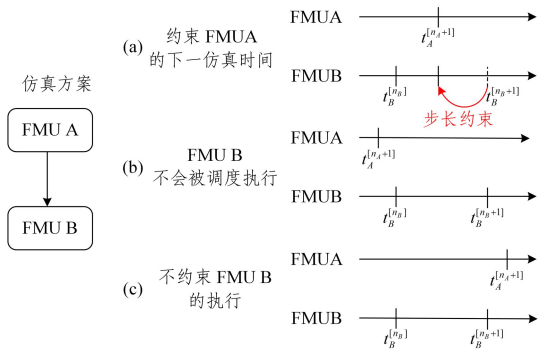

图 2 步长约束[22]

Fig. 2 Step size constraint[22]

1）基于输入的约束：假设 FMU A 的输入依赖于 FMU B 的输出，则 A 的下一仿真时刻 $t_A^{[n_A+1]}$ 不能超过 B 的下一仿真

时刻$t_B^{\lceil n_B+1\rceil}$。这是因为在$t_B^{\lceil n_B+1\rceil}$时，FMU A 应该更新输入为 FMU B 在$t_B^{\lceil n_B+1\rceil}$时刻产生的输出，以同步数据交换。

2)基于输出的约束：在所有依赖于 FMU A 的 FMU 到达 FMU B 的下一仿真时刻$t_A^{\lceil n_A+1\rceil}$之前，FMU A 不能被触发执行，以保证如果其他 FMU 没有推进，当前 FMU 不能更新其他 FMU 的输入，避免当前 FMU 过于领先，导致覆盖前一仿真时刻的输出。

3.3 同步和 API 互斥

由于 FMI API 的不可重入性，本文使用了缓冲区和读写锁实现 FMI 相关 API 的互斥访问。当其他 FMU 需要访问当前 FMU 的输入、输出、仿真时间时，从缓冲区中获取。具体而言，缓冲区以端口和值的映射缓存了 FMU 上一时刻的仿真输出。因此 FMU 可以持续推进仿真，避免了在相关 FMU 的仿真过程中，访问当前 FMU 的相关变量导致的不确定性。缓冲区同时缓存了 FMU 的下一仿真时间，该时间基于步长约束策略在相关 FMU 的调度判断阶段进行调整。此外，本文引入了累计计数策略以实现同步。仅在满足所有前置约束和后置约束的条件下，当前 FMU 才能继续调度。由于在主算法中，每个节点都进行前后及当前节点调度判断，因此一个依赖可能被判断两次。为了避免错误增加累计计数，主算法通过原子变量维护依赖判断的状态。

主算法使用了自定义的读写锁保护 FMU 的相关状态。在仿真执行设置状态时，FMU 的状态处于中间态，不允许其他线程进行仿真执行和调度判断，此时对 FMU 加写锁。而在调度判断时，不会修改 FMU 的状态，允许多个线程访问，此时对 FMU 加读锁。

3.4 并行优化

在每次仿真执行和调度判断时，需要通过读写锁保护 FMU 的资源。在主控算法中，临界区的执行时间较短，仅执行单次仿真步进或部分逻辑判断。而由此导致的加解锁会引起系统上下文的频繁切换，从而导致大量的 CPU 时间开销。此外，标准的读写锁不允许跨线程加锁和解锁，而算法中的加解锁操作需要由不同线程完成。具体而言，算法在新的仿真任务推送到调度器前由调度线程对 FMU 加锁，而解锁操作由执行线程完成。因此，本文自定义读写锁并使用 CAS (Compare and Swap)操作进行无锁并行优化。

本文定义的原子读写锁包含写原子变量 *WL*、读原子变量 *RL*，以及读者计数 *RC* 这 3 个参数。在调度判断前，主算法执行尝试获取读锁(Try Read Lock)操作，该操作首先对 *RL* 加锁，以保护读者计数。然后进行快速判断，当有其他读者时，不需要考虑是否有写者，仅增加读者计数。最后尝试对写锁加锁，当有线程在读时，其他线程不能对资源进行写操作。

在满足调度条件时，主算法对调度的 FMU 执行升级锁(Update Lock)操作。该操作首先对 *RL* 加锁，保护读者计数，并减少读者计数。然后尝试对写锁 *WL* 加锁，阻止后续读操作。该锁为写优先锁，可以避免写者饥饿问题。在解除读锁后进行忙等，直到没有其他读者并返回，此时其他线程不能再加锁。在主算法中，获取读锁的线程的读操作耗时较短，所以忙等时间也较短，不会导致过多的 CPU 开销。

4 FMI 并行联合仿真主算法

基于上述方法框架，本章给出 FMI 并行联合仿真主算法，主要包含预处理、初始任务调度，以及执行调度器 3 个阶段，如算法 1 所示。

算法 1 主算法

输入：FMU 实例列表 C，仿真方案图 G，线程数量 t

输出：预定义的 FMU 变量集合 Y

```
1. C̄ = Pretreat(C,G) // 预处理
2. P = Initialize(t) // 初始化调度器
3. for C_i ∈ C̄ do
4. DoSimulate(P,i,0) // 初始任务调度
5. end
6. WaitForFinish(P) // 等待调度器执行完成
```

4.1 预处理和初始任务

主算法的预处理阶段与 Van Acker 等[13]提出的方法类似。在仿真执行前，根据仿真方案配置，使用 Kosaraju 算法去除图中的强连通分量，生成 FMU 的有向无环图，然后使用拓扑排序，得到 FMU 的优先级列表。根据 3.2 节描述的步长约束，算法进行 FMU 步长和仿真时刻的初始化，将 FMU 的初始步长和仿真时刻初始化为同一环内 FMU 对应值的最小值。具体实现如算法 2 所示。

算法 2 仿真初始化

输入：已排序的 FMU 实例列表$\bar{C}$，FMU 的拓扑顺序优先级列表 R，变量的前置端口映射 Pre，FMU 实例的默认仿真步长$h_{i,0}$

输出：FMU 的步长列表 h，FMU 集合对应的下一仿真时刻 T

```
1. for C_i ∈ C̄ do
2.     h_i = min{h_j | R_i = R_j} // 环内最小值
3.     if h_{i,0} < h_i then
4.         h_i = h_{i,0} // 默认步长约束
5.     end
6. end
7. for C_i ∈ C̄ do
8.     T_i = h_i // 设置仿真时刻
9.     for j ∈ Pre(i) do
10.        if T_j < T_i then
11.            T_j = T_i // 前置依赖约束
12.        end
13.    end
14. end
```

与已有的 FMU 并行化算法不同，本文提出的线程池任务调度算法使用初始任务驱动后续仿真的执行，并通过调度判断代替全局同步和步长协商。因此，FMU 的后续仿真只受到前置和后置 FMU 的约束，并不受到其他 FMU 的影响，通过避免全局同步提供更好的并行性。虽然没有使用全局同步，但是受到步长约束的影响，每个 FMU 的仿真时间不会有很大的偏差，因此具有较好的实时性。

4.2 仿真任务

FMU 的一次仿真任务包括仿真执行和调度判断，如算法 3

所示。在仿真任务执行前，调度线程获取了执行仿真的 FMU 的写锁，避免了其他线程对该 FMU 缓冲的修改，保证了调度线程访问 FMU 的接口和缓冲的安全性。针对无输入输出依赖的 FMU，算法 3 中第 9－12 行快速进行后续调度的判断，避免了后续重复加解锁和初始化对应计数变量的开销。第 13 行初始化 FMU 的下一仿真时刻为当前仿真时刻与 FMU 的默认步长之和。当 FMU 受到后置节点的输入依赖时，将应用步长约束调整该值。第 16 行在仿真执行完成后释放 FMU 的写锁，以允许后续仿真的执行。第 17－24 行分别对前置节点、后置节点和当前节点进行调度判断，调度判断将在 4.3 节详述。

仿真任务将由调度器调度执行，因此需要考虑线程安全。线程安全需要满足以下两个同步约束：

1）在仿真执行阶段，只允许单个线程访问当前执行的 FMU 的相关资源，包括其输入缓冲区和 FMI API。

2）在调度判断阶段，允许多个线程访问判断的 FMU 的相关资源，但是需要保证相关操作的原子性。

针对约束 1，在 FMU 仿真执行前，由调度判断线程对该 FMU 加写锁，由仿真执行线程在执行完成后解锁。因此，对于 FMU 的同步约束，使用了自定义读写锁实现跨线程加解锁。对于跨线程加解锁的必要性解释如下：由于 FMU 的调度判断会被执行多次，但是对于该 FMU 的下一次，进只能由唯一线程执行一次。因此，如果由仿真执行线程对 FMU 加写锁，那么只要满足调度条件，该 FMU 的下一次仿真任务会被相关 FMU 多次推送到任务队列中，造成冗余并增加线程池负载。

算法 3 仿真任务(DoSimulate)

输入：FMU 的前置 FMU 依赖 Pre，FMU 的后置 FMU 依赖 Post，FMU 的变量缓冲映射 V(V_0 表示变量，V_1 表示值)，FMU 的下一仿真时刻缓冲 T_c，FMU 的前置依赖计数 CNT_{pre}，FMU 的后置依赖计数 CNT_{post}，FMU 的读写锁 LOCK，当前执行的 FMU C_i，当前仿真步长 h，仿真结束时刻 T_{end}

输出：预定义的 FMU 变量集合 Y

```
1. 仿真执行
2. for V∈V(i) do
3.     u:=V0 // 获取变量对应的端口
4.     v:=V1 // 获取变量对应的值
5.     fmiSe ti(u,v)
6. end
7. fmiDoStep(i,h) // 执行仿真
8. record(i,Y) // 记录当前 FMU 的输出
9. if Pre∈Ø and Post∈Ø and fmiGetTime(i)≤Tend then // 快速判断
10.    DoSimulate(i,h)
11.    return
12. end
13. Tc=fmiGetTime(i)+fmiGetStep(i)
14. CNTpre(i)=0 // 计数初始化
15. CNTpost(i)=0 // 计数初始化
16. WriteUnlock(LOCK(i))
17. 调度判断
18. for j∈Pre(i) do // 输入依赖处理
19.    CheckPre(j)
20. end
21. for j∈Next(i) do // 输出依赖处理
22.    CheckNext(j)
23. End
24. CheckCur(i) // 当前节点处理
```

4.3 调度判断

在当次仿真执行完成后，FMU 需要对前置、后置、当前 FMU 进行后续调度的判断。由于上述判断过程类似，此处以前置节点调度判断为例，描述算法细节及相关问题，前置节点调度判断如算法 4 所示。由于调度判断存在冗余，即 FMU 的前置节点、后置节点、FMU 本身都会对其进行调度判断，当无法获取对应 FMU 的读写锁时，表示有其他线程对该 FMU 进行调度判断。因此，在调度判断中，对于锁的获取使用非强制性策略。算法 4 中第 1－7 行分别获取前置 FMU 和当前 FMU 的读锁，以保护后续资源的访问。第 8－12 行对仿真时间进行判断，仅当前置 FMU 和当前 FMU 的仿真时刻相同时，表示前置 FMU 受当前 FMU 输入约束无法推进后续仿真，此时执行调度判断，否则释放锁。第 13－18 行进行 FMU 数据交换和步长约束。第 19－24 行进行调度判断，当满足以下 3 个条件时，FMU 可执行后续调度。

1）依赖计数和依赖 FMU 数量相同。当依赖计数和依赖 FMU 数量相同时，表示所有依赖都已处理，此时满足调度条件。

2）仿真时间小于结束时间。

3）成功获取对应 FMU 的写锁。Update 对已占有的读锁使用忙等，对已占有的写锁则直接返回失败。

算法 4 前置节点调度判断(CheckPre)

输入：端口映射 PM，其他输入符号定义同算法 3

输出：预定义的 FMU 变量集合 Y

```
1. if TryReadLock(LOCK(j))=false then // 访问 FMU 相关资源，获取读锁
2.     return
3. end
4. if TryReadLock(LOCK(i))=false then
5.     ReadUnlock(LOCK(j))
6.     return
7. end
8. if fmiGetTime(i)≠fmiGetTime(j) then
9.     ReadUnlock(LOCK(j))
10.    ReadUnlock(LOCK(i))
11.    return
12. end
13. for P∈PM(i,j) do
14.    fmiRea dj(Pout,V(i,Pin)) // 读取到缓冲
15.    Tc(j):=min{Tc(j),fmiGetTime(i)} // 步长约束
16.    CNTpost(j)? CN Tpost(j)+1
17.    ReadUnlock(LOCK(i))
18. end
19. if CNTpre(j)=size(Pre(j)) and
20. CNTpost(j)=size(Post(j)) and
21. fmiGetTime(j)≤Tend then
22.    r:=Upgrade(LOCK(j)) // 执行调度，获取写锁
```

```
23.   if r=true then
24.     DoSimulate(j,T_c(j)-fmiGetTime(j))
25.   end
26. end
```

5 案例演示

本章使用 Open Simulation Platform[23] 开源的房间温差模型(House)和船舶动态定位模型(Gunnerus-DP)验证提出的并行联合仿真方法的准确性和性能。测试环境为:具有16 GB RAM 以及 16 个“12th Gen Intel(R) Core(TM) i5-12500”逻辑处理器的计算机,在 FCST[24] 上运行。

5.1 实验设置

房间温差模型模拟了由两个房间组成的房子的温度偏差,表示房屋的热交换动力学。该模型的仿真方案由以下 FMU 组成:两个房间温度变化模型、内部墙壁热传导模型、两个外部墙壁热传导模型、温度控制器模型,以及时钟重置信号模型。其输入和输出端口依赖关系如图 3 所示。将每个 FMU 视为黑盒,观察两个房间的温度变化,即 Room1 和 Room2 的 T_room 端口的输出。

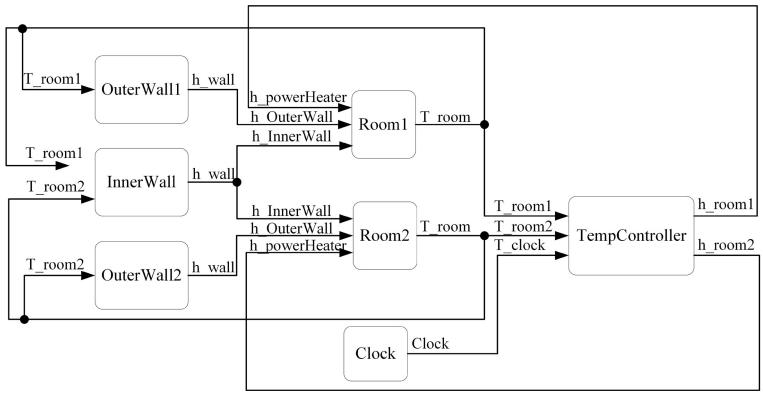

图 3 房间温差模型仿真方案

Fig. 3 Simulation scheme for House

船舶动态定位模型模拟了一艘动态定位(Dynamic Positioning,DP)控制的船舶(Gunnerus)在受到外力时的箱体移动。仿真方案由以下 FMU 组成:箱体定位参考模型、产生表面流速的海流模型、三自由度 DP 控制器、为 DP 提供定位点的参考模型、发动机动力、六自由度船舶模型。模型间的输入和输出端口依赖关系如图 4 所示。

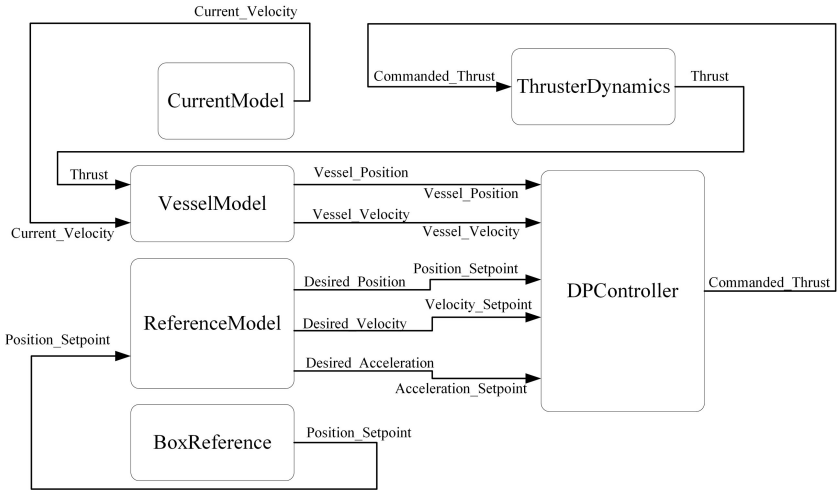

图 4 船舶动态定位仿真方案

Fig. 4 Simulation scheme for Gunnerus-DP

将本文提出的方法称为 Pool-Cosim(基于线程池调度的局部同步联合仿真方法),与以下方法的仿真性能进行比较,观察仿真精度和仿真速度。

1)使用单线程的非迭代雅可比定步算法[25](NI-Jacobi)。该方法使用经典 FMI 联合仿真主算法,是最广泛使用的联合仿真方法。该方法作为数值基准,用于观察 Pool-Cosim 的准确性。

2)使用 FMU 并行化[5]的非迭代雅可比定步算法(P-NI-Jacobi)。该方法作为基准,对比 Pool-Cosim 的性能。

5.2 准确性测试

由于本文方法主要考虑性能,未在精度方面进行改进,因此仿真结果应与标准方法相同或相近。将 House 模型的联合仿真输出结果与参考输出结果进行对比,验证所提出方法联合仿真的数值结果。由于 FMU 的封装性,无法解析 FMU 的方程,因此参考输出结果由 NI-Jacobi 获得,文献[25]已给出 NI-Jacobi 很好的数值精度。设置仿真时间为 100 s,每个

模型的仿真步长为 0.01 s。图 5 显示了房间温度的仿真结果，仿真结果与 NI-Jacobi 相符。对于联合仿真的不同输出，也得到了类似的精度结果。

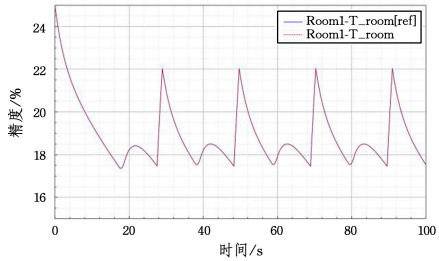

图 5 Pool-Cosim 准确性测试

Fig. 5 Accuracy test for Pool-Cosim

5.3 性能测试

实验使用 Boost 1.86 的 boost::asio::thread_pool 作为 Pool-Cosim 的调度器所使用的线程池进行测试，Boost Asio 被广泛应用于高性能、低延迟的网络编程和分布式系统中，其稳定性和性能已被实际使用和证明。为分析 Pool-Cosim 方法的并行性能，使用 1—8 个线程测试算法的仿真耗时、加速比和 CPU 使用率，并与 NI-Jacobi 和 P-NI-Jacobi 对比。仿真配置 House 方案的仿真时间为 2 000 s，模型步长为 0.01 s；Gunnerus-DP 方案的仿真时间为 2 000 s，模型步长为 0.1 s。每个场景和算法组合均重复 50 次仿真，以评估算法的稳定性。仿真结果性能对比如表 1 和表 2 所列，等线程和最优线程下的仿真耗时统计数据对比如图 6 和表 3 所示。为了更清楚地展示仿真耗时，图 7 和图 8 进一步给出了仿真耗时随线程变化的对比。

表 1 House 性能对比

Table 1 Performance comparison test for House

算法	线程数	平均耗时/s	平均 CPU 使用率/%	加速比
NI-Jacobi	1	31.076	6.12	1.00
P-NI-Jacobi	8	34.228	26.75	0.91
Pool-Cosim	1	52.857	6.31	0.59
Pool-Cosim	2	29.523	12.05	1.05
Pool-Cosim	**3**	**22.785**	**17.18**	**1.36**
Pool-Cosim	4	23.402	17.82	1.33
Pool-Cosim	5	25.865	15.67	1.20
Pool-Cosim	6	26.912	15.89	1.15
Pool-Cosim	7	28.114	15.88	1.11
Pool-Cosim	8	28.938	15.95	1.07

表 2 Gunnerus-DP 性能对比

Table 2 Performance comparison test for Gunnerus-DP

算法	线程数	平均耗时/s	平均 CPU 使用率/%	加速比
NI-Jacobi	1	14.204	5.86	1.00
P-NI-Jacobi	7	10.495	19.50	1.35
Pool-Cosim	1	17.062	5.93	0.83
Pool-Cosim	2	10.422	11.01	1.36
Pool-Cosim	3	9.511	12.23	1.49
Pool-Cosim	**4**	**9.468**	**12.74**	**1.50**
Pool-Cosim	5	9.616	12.43	1.48
Pool-Cosim	6	9.641	12.86	1.47
Pool-Cosim	7	9.844	12.90	1.44
Pool-Cosim	8	9.763	12.87	1.45

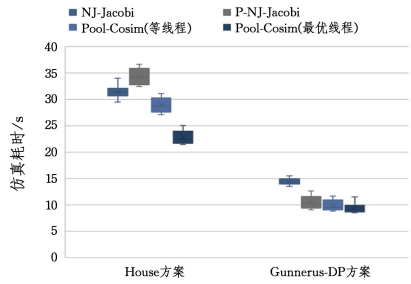

图 6 仿真耗时统计对比

Fig. 6 Comparison of simulation time cost statistics

表 3 仿真耗时标准差和 95% 置信区间(t 分布)误差范围

Table 3 Simulation time SD and 95% CI(t-distribution) MoE

仿真方案	算法	线程	SD	MoE
House	NI-Jacobi	1	1.25	0.89
House	P-NI-Jacobi	8	1.52	1.09
House	Pool-Cosim	8	1.51	1.08
House	Pool-Cosim	3	1.28	0.91
Gunnerus	NI-Jacobi	1	0.62	0.44
Gunnerus	P-NI-Jacobi	7	1.23	0.88
Gunnerus	Pool-Cosim	7	1.06	0.76
Gunnerus	Pool-Cosim	4	1.08	0.77

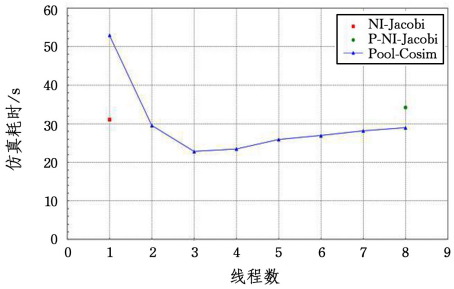

图 7 House 仿真耗时对比

Fig. 7 Performance comparison test for House

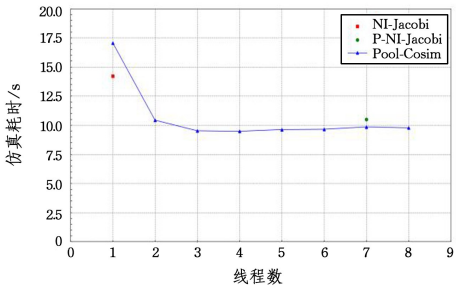

图 8 Gunnerus-DP 仿真耗时对比

Fig. 8 Performance comparison test for Gunnerus-DP

以 House 方案为例，对 Pool-Cosim 仿真耗时变化趋势进行分析。在线程数为 3 时，Pool-Cosim 达到了最短的仿真耗时以及最高的加速比。由于参与仿真的 FMU 数量只有 7 个，并且同一 FMU 在同一时刻只能由单一线程调度，因此同一时刻执行仿真任务的线程数量最多为 7。当线程数量大于 3 时，线程间同步以及竞争访问任务队列的性能开销高于多线程并行处理任务的性能提升，所以仿真性能有所下降。同时，由于算法并发性存在上限，当多线程竞争获取任务时，无法获取到任务的线程会处于阻塞状态，导致线程无法充分利

用 CPU，进而导致 CPU 使用率下降。

由于仿真方案的加速比上限受仿真步长、模型数据依赖关系的影响，因此，不同仿真场景及不同步长配置下，算法的性能提升存在差异。在 House 方案（步长为 0.01）中，由于仿真步长较小，体现出了 P-NI-Jacobi 的劣势，线程同步的开销高于其所带来的性能提升，其加速比仅为 0.91，低于单线程的 NI-Jacobi 方法。而 3 线程的 Pool-Cosim 表现最佳，加速比达到 1.36。在 Gunnerus-DP 方案（步长为 0.1）中，Pool-Cosim 相比于 P-NI-Jacobi 也表现出更好的性能。从加速比等比提升方面考虑，P-NI-Jacobi 相比串行方法仿真速度提升了 35%，而 Pool-Cosim 在等线程下相比串行方法提升了 44%，该比例相对于 P-NI-Jacobi 提升了 25.7%。Pool-Cosim 在最优线程下相比串行方法提升了 50%，该比例相对于 P-NI-Jacobi 提升了 42.9%。综上，Pool-Cosim 在最优线程下表现出显著的性能提升。同时，在等线程条件下，Pool-Cosim 的仿真耗时和加速比也优于 P-NI-Jacobi 方法。

从统计数据分析也可以看出，Pool-Cosim 算法具有较好的稳定性。此外，P-NI-Jacobi 表现出较高的 CPU 使用率，Pool-Cosim 在各线程下的 CPU 使用率均优于 P-NI-Jacobi 方法。因此，Pool-Cosim 在消耗更少的系统资源的同时具有更好的仿真性能。

5.4 消融实验

针对 3.2 节提出的调度器和 3.4 节提出的并行优化方法，进行消融实验，以观察线程池和无锁优化的性能提升。对比方法如下。

1）Local-Cosim：本文提出的局部同步的并行优化算法，使用 FMU 并行化。

2）Pool-Cosim：在 Local-Cosim 方法上使用 Boost 库进行线程池并行优化。

3）Pool-Cosim-Lock（PCL）：对于 Pool-Cosim 方法中的同步问题，使用阻塞锁。

4）Pool-Cosim-Lockfree（PCLF）：对于 Pool-Cosim 方法中的同步问题，使用 3.4 节提出的非阻塞读写锁。

对比 Local-Cosim 和 Pool-Cosim，以观察线程池优化的性能提升。使用 5.1 节的房间温差模型对 Local-Cosim 进行性能测试，结果如表 2 所列。

表 2 Local-Cosim 性能测试结果

Table 2 Performance test for Local-Cosim

算法	线程数	时间/s	平均 CPU 使用率/%	加速比
Local-Cosim	7	28.68	20.85	1.08

相比于 Pool-Cosim，在等线程条件下，Local-Cosim 与 Local-Cosim 的仿真耗时接近，但是平均 CPU 使用率较高。而在 Pool-Cosim 使用 3 线程时，线程池优化表现出显著的性能提升。对比 P-NI-Jacobi，Local-Cosim，Pool-Cosim（3 线程）的同步计算 CPU 占比，结果如图 9 所示。由图 9 可以看出，Pool-Cosim 方法的同步开销远低于 P-NI-Jacobi 和 Local-Cosim 方法，更充分地利用了 CPU，并缩短了仿真耗时。

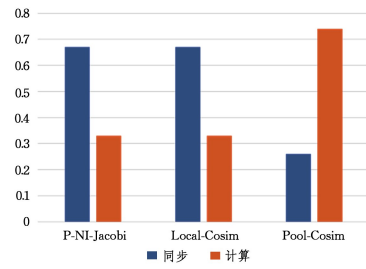

图 9 房间温差模型同步计算 CPU 占比

Fig. 9 CPU ratio for synchronization and computation of House

对比 Pool-Cosim-Lock（PCL）和 Pool-Cosim-Lockfree（PCLF），以观察无锁优化的性能提升。对仿真耗时和锁相关 CPU 时间进行测试，结果如表 3 所列。

表 3 无锁优化仿真耗时和同步耗时对比

Table 3 Comparison of simulation time for lock free optimization

(s)

线程数	PCL 仿真耗时/s	PCLF 仿真耗时	PCL 同步耗时	PCLF 同步耗时
1	53.830	52.857	3.392	2.052
2	30.230	29.523	4.173	2.280
3	**24.331**	**22.785**	**4.928**	**2.501**
4	24.853	23.402	3.998	1.922
5	27.044	25.865	4.038	1.732
6	29.130	26.912	2.936	1.436
7	29.543	28.114	2.230	1.231
8	29.563	28.938	1.799	1.040

相比 Pool-Cosim-Lock，Pool-Cosim-Lockfree 的性能提升在于优化了同步耗时，即锁相关 API 的 CPU 消耗。由实验结果可以看出，对于联合仿真这类小任务频繁进行数据交换的场景，无锁优化大幅减少了同步的 CPU 消耗。

5.5 结果分析

Pool-Cosim 在保持联合仿真精度的情况下，显著提高了仿真性能。该方法允许自动调度 FMU 的仿真执行，而不必手动为 FMU 分配执行线程，因此也不需要关心 FMU 的单步执行时间和依赖关系。然而，Pool-Cosim 方法的效率提升基于选择了合适的线程数量，线程数量需要根据仿真方案中 FMU 的数量以及机器配置进行选择。线程数量较少时，无法充分发挥算法的并行性，而线程数量较多时，则会增加同步的性能开销。针对线程数量，通过探究多个仿真方案的实验结果，本文总结出以下启发式规则：

1）线程数量根据仿真方案图中 FMU 拓扑顺序导致的并行性决定。有拓扑先后顺序关系的 FMU 在单步仿真执行完成后需要等待其他依赖的 FMU 的仿真执行，因此单步推进时间取决于这些 FMU 中单步仿真执行时间最长的 FMU。因此，对于仿真执行时间差距较大且有拓扑先后顺序的 FMU，需要使用少于 FMU 数量的线程。

2）线程数量需要根据参与仿真的 FMU 数量决定，且参与仿真的 FMU 数量增加，最佳线程数量增加，但增长率降低。对于平均单步仿真执行时间相同的 FMU，线程数量越多，仿真单元执行时间的不均匀性越大，此时线程池任务调度的优势更加明显，先执行完一个 FMU 仿真的线程可以执行

其他 FMU 的仿真，避免阻塞等待。

3)线程数量根据机器 CPU 核心数量决定，超过 CPU 核心数量会导致性能下降，这是由并行计算的规则决定的。

在 FCST[24]中，将线程数量作为该方法的一个可配置参数，以提供更灵活的仿真配置。

结束语 本文提出了基于线程池任务调度的局部同步 FMI 并行联合仿真方法。该方法在房间温差模型和船舶定位模型上进行了验证，结果表明，Pool-Cosim 在保持仿真精度的同时在仿真性能上有显著提升。

本文的研究面向单机联合仿真场景。针对分布式仿真的扩展性，本文提出的局部同步和无锁并行优化方法可以应用于单节点上的多个仿真单元间的数据交互和仿真运行，以加速单节点仿真。此外，本文提出的任务封装策略也可以扩展至分布式架构，将仿真同步约束在从节点之间，避免全局同步的开销，降低主节点的负载，因此，在未来的研究工作中，将探索 Pool-Cosim 在分布式仿真场景下的扩展实现。此外，由于 FMI 标准面向模型固定仿真场景，在未知模型实时仿真方面存在局限性。因此，未来也将探索 FMI 3.0 标准的特性在 FMI 并行仿真中的应用，针对联合仿真中的事件建立同步和回滚机制。同时，结合 FMI 联合仿真标准与高级体系结构(High Level Architecture，HLA)等实时仿真技术，以适应更复杂的分布式联合仿真场景。

参 考 文 献

[1] HATLEDAL L I，STYVE A，HOVLAND G，et al. A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface[J]. IEEE Access，2019，7：109328-109339.

[2] BLOCHWITZ T，OTTER M，ARNOLD M，et al. The Functional Mockup Interface for Tool independent Exchange of Simulation Models[C]// Proceedings of the 8th International Modelica Conference. Dresden：Linköping University Press，2011：105-114.

[3] GALTIER V，VIALLE S，CHERIFA D，et al. FMI-based distributed multi-simulation with DACCOSIM[C]// Proceedings of the Symposium on Theory of Modeling & Simulation：DEVS Integrative M&S Symposium. 2015：39-46.

[4] ÉVORA GÓMEZ J，HERNÁNDEZ CABRERA J，TAVELLA J P，et al. Daccosim NG：co-simulation made simpler and faster [C]// The 13th International Modelica Conference. 2019：785-794.

[5] DAD C，TAVELLA J P，VIALLE S. Synthesis and feedback on the distribution and parallelization of FMI-CS-based co-simulations with the DACCOSIM platform[J]. Parallel Computing，2021，106：102802.

[6] KRAFT J，MEYER T，SCHWEIZER B. Parallel Co-Simulation Approachwith Macro-Step Size and Order Control Algorithm [C]// 15th International Conference on Multibody Systems，Nonlinear Dynamics，and Control. 2019.

[7] POGORELOV D，RODIKOV A，KOVALEV R. Parallel Computations and Co-simulation in Universal Mechanism Software. Part I：Algorithms and Implementation[J]. Transport Problems，2019，14(3)：163-175.

[8] BEN KHALED A，BEN GAID M，PERNET N，et al. Fast multi-core co-simulation of Cyber-Physical Systems：Application to internal combustion engines[J]. Simulation Modelling Practice and Theory，2014，47：79-91.

[9] SAIDI S E，PERNET N，SOREL Y. Automatic parallelization of multi-ratefmi-based co-simulation on multi-core[C]// TMS/DEVS 2017－Symposium on Theory of Modeling and Simulation. ACM，2017.

[10] SAIDI S E，PERNET N，SOREL Y. A method for parallel scheduling of multi-rate co-simulation on multi-core platforms [J]. Oil & Gas Science and Technology－Revued'IFP Energies Nouvelles，2019，74：49.

[11] HANSEN S T，GOMES C，KAZEMI Z. Synthesizing Orchestration Algorithms for FMI 3.0[C]// 2023 Annual Modeling and Simulation Conference(ANNSIM). 2023：184-195.

[12] BROMAN D，BROOKS C，GREENBERG L，et al. Determinate composition of FMUs for co-simulation[C]// 2013 Proceedings of the International Conference on Embedded Software(EMSOFT). IEEE，2013：1-12.

[13] VAN ACKER B，DENIL J，VANGHELUWE H，et al. Generation of anOptimised Master Algorithm for FMI Co-simulation [C]// Proceedings of the Symposium on Theory of Modeling & Simulation：DEVS Integrative M&S Symposium. 2015：205-212.

[14] CREMONA F，LOHSTROH M，TRIPAKIS S，et al. FIDE：an FMI integrated development environment[C]// Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM，2016：1759-1766.

[15] WU Q，COLE C，SPIRYAGIN M，et al. Parallel co-simulation of heavy-haul train braking dynamics with strong nonlinearities [J]. Mechanics Based Design of Structures and Machines，2024，52(5)：2623-2638.

[16] SHEN Y，LIU C，ZHOU D，et al. A parallel coupling framework for DEM-MBD：Model verification and application DEM-MBD [J]. Powder Technology，2024，448：120257.

[17] ZHENG L，CUI Y，JIN S，et al. High-Performance Computing-Based Open-Source Power Transmission and Distribution Grid Co-Simulation[J]. IEEE Transactions on Power Systems，2024，39(5)：6144-6153.

[18] STEWART R，RAITH A，SINNEN O. Optimising makespan and energy consumption in task scheduling for parallel system [J]. Computers & Operations Research，2023，154：106212.

[19] LI M，HUANG L，XU G，et al. A parallel particle swarm optimization framework based on a fork-join thread pool using a work-stealing mechanism[J]. Applied Soft Computing，2023，145：110537.

[20] FATOUROU P，KOSMAS E，PALPANAS T，et al. FreSh：A Lock-Free Data Series Index[C]// 2023 42nd International Symposium on Reliable Distributed Systems(SRDS). 2023：209-220.

[21] MAROTTA R，IANNI M，PELLEGRINI A，et al. A Conflict-

Resilient Lock-Free Linearizable Calendar Queue[J]. ACM Transactions on Parallel Computing, 2024, 11(1): 1-32.
[22] EGUILLON Y, LACABANNE B, TROMEUR-DERVOUT D. F$ $_3$ $ORNITS: a flexible variable step size non-iterative co-simulation method handling subsystems with hybrid advanced capabilities[J]. Engineering with Computers, 2022, 38(5): 4501-4543.
[23] PERABO F, PARK D, ZADEH M K, et al. Digital Twin Modelling of Ship Power and Propulsion Systems: Application of the Open Simulation Platform(OSP)[C]// 2020 IEEE 29th International Symposium on Industrial Electronics(ISIE). IEEE, 2020: 1265-1270.
[24] LU L, XUE Z, LIU F. Design and Development of an FMI Co-Simulation Tool[C]// 2024 3rd International Conference on Artificial Intelligence and Computer Information Technology(AICIT). 2024: 1-5.
[25] SICKLINGER S, BELSKY V, ENGELMANN B, et al. Interface Jacobian-based Co-Simulation[J]. International Journal for Numerical Methods in Engineering, 2014, 98(6): 418-444.

XUE Zhaoyang, born in 1999, postgraduate, is a member of CCF (No. Y8623G). His main research interest is modeling and simulation.

LIU Fei, born in 1976, Ph.D, professor, Ph.D supervisor, is a member of CCF (No. B9231M). His main research interests include modeling and simulation, and artificial intelligence.

(责任编辑:柯颖)

2025 年度会员发展贡献 TOP10 揭晓

CCF 作为全国性学会,以“为计算领域的专业人士服务”为宗旨,吸纳了众多专业人士加入学会,会员通过自己对学会的了解和认同,推荐计算领域专业人士加入学会,让学会不断发展、壮大。

2025 年有 1981 位会员为 CCF 推荐了 10478 位新会员,其中贡献突出的 TOP10 会员名单如下:

1)数据来源:CCF 会员推荐会员系统

2)统计时间:2025 年 1 月 1 日—2025 年 12 月 31 日

3)按推荐会员总数排序(4 个学生会员折算 1 个专业会员)

据 CCF 微信公众号