CHAPTER 5

第5章　分布式混合仿真的时间管理研究

5.1　引言

时间是分布式混合仿真的核心概念和重要基础，时间管理作为仿真引擎的核心功能，一直是学术界研究的热点和难点。在分布式仿真系统中，存在地理分散、结构各异的仿真节点和实体，它们的仿真时钟也不尽相同。时间管理的目的就是在同一个仿真时间轴上推进仿真实体运行，保证仿真事件遵守正确的因果逻辑，从而保证仿真结果的正确性。时间同步是时间管理的基础功能，目前主要有两种方式[108]：硬件同步是采用硬件手段比如高精度时钟或 GPS 授时等硬件设备，对不同平台进行统一授时达到同步的效果；软件同步则依靠相关的软件同步策略，保证仿真事件按照正确的逻辑顺序来运行。本章主要基于软件同步的方式对时间管理进行研究，将从时间同步策略和仿真事件调度优化两方面展开论述。

5.2　时间同步策略

如本书 4.3.1 节所述，面向装备智能化保障体系的分布式混合仿真将智能

体、离散事件仿真都转换为基于 DEVS 的离散事件仿真规范实现，因此混合仿真的时间管理策略实际可转换为基于 DEVS 模型的离散事件仿真时间同步问题。本书中，DEVS 模型与离散事件仿真系统一样，遵守自己的仿真时间系统，在事件发生或更新时进行同步。仿真系统负责对各个模型的时间基准进行校准[79]。因此，在仿真运行的过程中，一是要确保以相同的逻辑时钟对仿真事件进行标记，二是要保证所有的 DEVS 模型按照适合的时间同步策略处理仿真推进的问题。

第一个问题可通过设置全局-本地时间基准服务器来解决[109]。整个系统可以按照网络部署，在每个分块的局域网区域配置一个本地时间基准服务器，本地时间基准服务器先与全局服务器进行对时，再向下提供时间基准服务[79]。

第二个问题可通过仿真事件时空调度方法进行[79]。主要有两类办法[110]，一类是在仿真事件调度过程中严格按照事件的时戳大小来处理，这样不会产生因果错误，即保守策略[111]；另一类是采用各仿真实体模型先分布式运行，发生时序错误时再进行相应处理，即乐观策略[112]。

5.2.1　保守的时间同步策略

保守的时间同步[113]策略是以仿真的离散事件保守算法为基础，实现仿真系统的可实现性及可预测性。在保守策略中，“时间前瞻量”(Lookahead)和“时戳下限”(Lower Bound Time Stamp，LBTS)是影响时间同步的两个关键参数，引入“时间前瞻量”可以有效解决死锁的问题[114]。

1. 时间前瞻量[115]

设仿真模型的逻辑时钟当前为 T，并且这个仿真实体通过计算预测出下一事件的发生不会超过 $T+L$，则称 L 就是这个仿真实体计算出的“时间前瞻量”。它表明该仿真实体再不会在未来的 Lookahead 时间内产生新的事件。

2. 时戳下限值[116]

时戳下限值表示仿真实体不会收到时戳小于 LBTS(i)的事件，该值由仿

真引擎通过轮询所有实体计算而得，如图 5.1 所示。

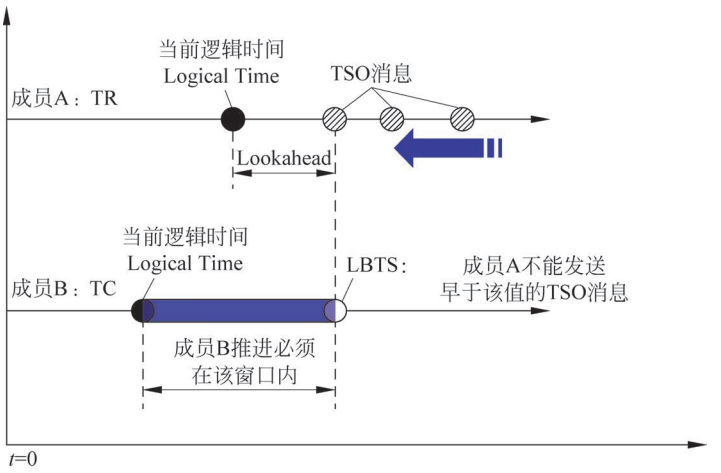

图 5.1　仿真成员的时间推进图

仿真成员 LBTS 的计算公式如下[118]：

$$\mathrm{LBTS}_i = \mathrm{Min}(T_c(j) + \mathrm{Lookahead}(j)) \tag{5-1}$$

其中，T_c 为仿真实体的当前时间；Lookahead(j)是其 Lookahead 值。仿真节点的 LBTS 定义为该节点中所有仿真实体 LBTS(i)的最小值，即 LBTS=Min{$LBTS(i)$}。

有效逻辑时间(Effective Logical Time，ELT)[119]：仿真实体的 ELT(i)表示该实体可以发送的 TSO 消息时间戳的下限值，如图 5.2 所示。由 Lookahead 的定义可知

$$\mathrm{ELT}_i = T_c(i) + \mathrm{Lookahead}(i) \tag{5-2}$$

其中，$T_c(i)$为仿真实体的当前逻辑时间。由于 Lookahead 和 ELT 都由仿真实体自己计算得到，它表示仿真实体对自己未来事件的预测。仿真节点的 LBTS 是仿真引擎根据各个仿真实体的 ELT 计算而得，它决定了各仿真成员逻辑时钟的推进。因此，保守的时间同步策略就是围绕计算 LBTS，并在此基础上进行调度和优化实现的[120]。

在保守策略算法[120]中，仿真成员总是以递增的时戳顺序发送事件，这样可以基本上确保目标实体事件队列的时戳是递增的。仿真进程处理输入事件时，只有在收到时戳大于当前时间的事件后才开始执行，否则就一直等待。但是这

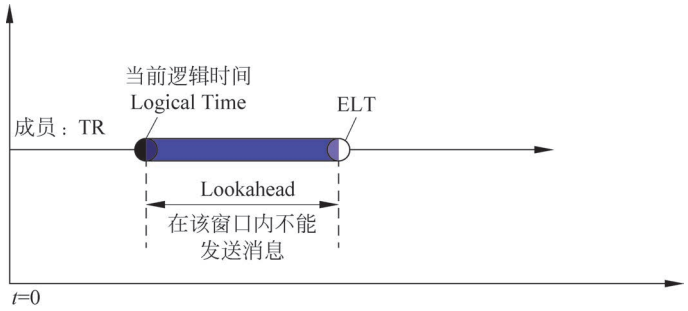

图 5.2　仿真成员的 ELT 示意图

种方法在没有输入事件到达时将引起死锁，导致进程一直处于等待状态。通常的解决方法是在事件队列为空时，就发送一条不包含任何内容的空消息去推进仿真，从而避免引起死锁。其中空消息也是带时戳的，其值是 T＋Lookahead。针对保守策略算法，已有大量的研究成果，如 CMB 算法[121]、Misra 算法[122]、保守时间窗算法[123]等，本书不再赘述。

5.2.2　乐观的时间同步策略

乐观的时间同步策略要比保守的时间同步策略复杂很多，它允许因果错误发生，但是在因果错误发生时，仿真引擎应能够检测到错误并将仿真回滚到错误发生之前的状态。相对保守策略而言，乐观的时间推进与具体的仿真模型无关，并发性较好；而保守策略则一定程度上依赖具体的仿真应用，特别是 Lookahead 的确定。乐观策略主要包括仿真模型的状态监控、状态回滚等操作，回滚机制由一定的事件来触发，例如收到一个 Straggler 事件或者反事件[124]。

Straggler 事件[125]：仿真实体收到了时戳不按递增方式增加的异常事件。

反事件[126]：一个条件触发事件，如果收到了该事件，仿真引擎会立即进行状态回滚，并撤销之前已执行的事件。

回滚需要 DEVS 模型在运行过程中监控和保存其状态信息，因此需要专门的状态队列来保存这些数据，另外还需要设置输入事件和输出事件队列来保存

之前的内部和外部事件数据。仿真引擎通过调度这些队列推动仿真运行。这也意味着乐观策略在快速推进仿真的同时，需要付出额外的成本来保存和恢复每个进程的运行状态。对于回滚的相关机制，主要有懒惰取消（Lazy Cancellation）[127]、乐观时间窗[128]、懒惰再评估（Lazy Revalution）[129]、懒惰回滚（Lazy Rollback）[130]等，这里也不再赘述。

5.2.3 混合的时间同步策略

相关文献作出了保守策略和乐观策略对比结果，如表 5.1 所示。

表 5.1 保守策略与乐观策略对比[131]

对比内容	保守策略	乐观策略
实现原理	不允许因果错误发生，严格按时戳的大小顺序处理仿真事件	各逻辑进程各自推进，检测到因果错误后进行回滚操作
推进策略	利用带时戳的空消息来推进时间	各自推进，遇错回滚
前瞻量计算	需要	不需要
全局虚拟时间	不需要	需要
事件处理	各 LP 自建输入队列，严格按时戳顺序执行	不必按 FIFO 规则接收事件，需频繁对事件队列进行插入或排序
实现方式	控制较为简单，数据结构主要以堆栈为主	控制较为复杂，需要对状态队列进行管理，对事件进行调度
性能表现	依赖前瞻量计算和死锁管理策略，事件平均计算和通信所需的开销较小	依赖模型状态存储空间的处理与回滚机制，以及事件调度检索，通信开销较高

由于保守的时间同步策略其仿真性能严重依赖前瞻量的计算，而在同一个节点内其前瞻量相对容易确定，节点内的网络拓扑结构也相对稳定，再加上面向装备智能化保障体系的仿真系统是一个分布式系统，部署在不同的节点上运行。因此可以采用保守策略和乐观策略相结合的方法，在节点内部严格按照仿真事件时戳的因果顺序执行，在节点间采用遇错回滚的方法，同时在仿真节点中设置局部和全局时间同步服务器进行对时[79]，如图 5.3 所示。

从图 5.3 中可以看出，每个节点上的 DEVS 模型可以组合在一起，形成一

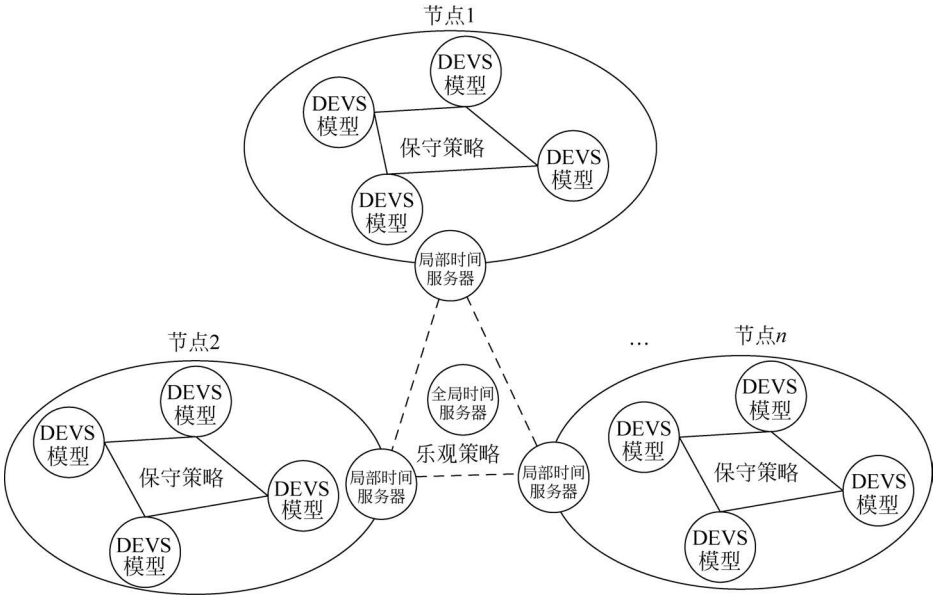

图 5.3　混合的仿真时间同步策略

种内部的时间同步方式。由于组内的仿真模型在计算 Lookahead 时更容易计算出来，可以有效避免网络中空消息的发送，进而减小系统开销和复杂度。在节点间采用乐观策略，仿真节点可以不再等待空消息来推进，能够大大提高仿真的并发性[131]。

5.3　基于自平衡二叉排序树的仿真事件调度优化方法

在对仿真时间管理的研究中，除了时间推进机制外，还存在一些特别影响仿真运行效率的因素，比如下文将要详细论述的仿真事件队列调度，也同样需要优化。事件队列是 DEVS 引擎的基本元素，离散事件调度算法保证仿真的因果性和时序性。对于 DEVS 分布式并行仿真框架而言，每个 LP 内部和中心服务器都要采用离散事件调度算法对仿真中产生的事件进行调度。由于事件队列的排序是调度算法的核心操作，所以对事件的排序进行优化也是提升 DEVS 仿真性能的一个重要手段。目前 DEVS 仿真引擎中采用自排序队列存储仿真

事件，但仿真中事件队列是在动态变化的。因此查找、插入和弹出等各种操作的时间复杂度都是不稳定的，无法发挥自排序队列的优势。二叉排序树是一种插入式排序算法的容器，它的操作时间复杂度最好情况是 $O(\lg n)$。而且在二叉排序树基础上发展起来的自平衡二叉排序树（AVL 树）不分好或坏的情况，其时间复杂度均保持 $O(\lg n)$ 不变。因此 AVL 排序树是一个非常理想的插入式排序队列的容器。下面首先介绍二叉排序树的原理和 AVL 排序树，随后详细阐述保守策略和乐观策略两种时间同步方法中基于 AVL 排序树的事件调度算法。

5.3.1 仿真事件排序优化方法

采用中序的方式对二叉树进行遍历，能够将节点值按照大小进行排序，构造出一个有序的二叉树。因此，可以采用这种方法将一个无序的序列变为一个有序的二叉排序树，每次插入新的节点，都是在叶子节点上进行操作，无须移动其他节点。搜索、插入和删除的复杂度与树的深度相关，期望为 $O(\lg n)$，最坏的情况是 $O(n)$。下面简要介绍二叉排序树的插入和删除操作，其结构如图 5.4 所示。

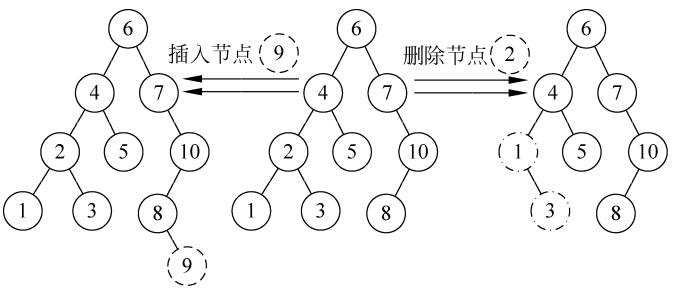

图 5.4 二叉排序树的插入和删除操作

图 5.4 给出了一个二叉排序树插入和删除节点的例子。其中，删除操作节点 2 的操作为首先将节点 2 删除，再将节点 1 挂到节点 4 上，最后把节点 3 挂到节点 1 上。

AVL 排序树，其实就是二叉搜索树中的深度自平衡排序树，任意节点的左

右两个子树的深度之差小于 1，该特性称作深度平衡[134]。节点的左右子树深度差就是该节点的平衡因子。当所有节点的平衡因子绝对值不大于 1 时，AVL 排序树就是平衡的。一旦节点的平衡因子超出范围，就需要对树进行旋转操作以回到平衡状态。下面通过一个定理说明自平衡性的重要性。

定理 5.1：一棵拥有了 n 个元素的 AVL 排序树的深度为 $O(\lg n)$。

证明：令 $n(h)$ 为一棵深度为 h 的 AVL 排序树的最小节点数，可以得到 $n(1)=1$ 且 $n(2)=2$。

当 $n>2$ 时，一棵深度为 h 的 AVL 排序树包含一个根节点，一个深度为 $h-1$ 的 AVL 子树和另一个深度为 $h-2$ 的 AVL 子树。因此，$n(h)=1+n(h-1)+n(h-2)$。

已知 $n(h-1)>n(h-2)$，使 $n(h)>2n(h-2)$。因此有 $n(h)>2n(h-2)$，$n(h)>4n(h-4)$，$n(h)>8n(h-6)$，…，$n(h)>2^i n(h-2i)>2^{|h/2|-1}(1)=2^{|h/2|-1}$。

得到 $n(h)>2^{h/2-1}$。

经过对数运算，有 $h<2\lg n(h)+2$。

故命题得证，AVL 排序树的深度为 $O(\lg n)$。

AVL 排序树的深度和操作的时间复杂度一致，所以能够通过旋转将非平衡态的树变换成平衡态的 AVL 排序树，从而减小树的深度，达到降低操作复杂度的目的。

AVL 排序树的时间复杂度非常稳定，不管什么情况下进行插入、删除等操作，其时间复杂度都为 $O(\lg n)$。相比于普通的二叉排序树，AVL 排序树更稳定。下面简要介绍 AVL 排序树的插入和删除操作后的自旋情况。

插入一个节点后，检查节点的子树是否还符合 AVL 排序树的规则，如果每个节点平衡因子仍然处于{−1,0,1}范围中，则说明不需要进行旋转。但是，如果平衡因子超出范围，则说明这个节点的子树不平衡，则需要一次或两次旋转以保证 AVL 排序树的平衡性。存在 4 种情况，且这 4 种情况两两对称[133]。

5.3.2　仿真事件队列结构优化方法

根据 DEVS 仿真的特性，原子模型的事件有三个特点：①原子模型输入事

件有内部状态迁移事件和外部输入事件两种类型；②内部状态迁移事件的时戳为下一次模型进行状态迁移的时刻；③外部输入事件的时戳一般都与其输出逻辑顺序保持一致，所以输入事件只需要确定与内部状态迁移事件的先后顺序即可，基本不需要排序。

根据这三个特点，可以将原子模型的输入事件队列近似看作一个自有序的FIFO队列，其排序的时间消耗很小。因此，本节摒弃以往离散事件引擎中将所有事件集中处理的方式，为每个原子模型构造一个输入事件队列。每个原子模型同时最多只能有一个时戳最小的事件进入全局事件队列。对全局事件队列排序以维护仿真中所有事件的有序性。一旦某事件的时戳成为全局最小时戳，就将其弹至相应原子模型的输入端口。这种两层事件队列结构，使全局事件队列的长度与实体的数目相等，不会产生剧烈的变化。值得注意的是，由于原子模型的内部状态迁移事件也参与排序，这就保证了全局事件队列弹出事件的时戳全局最小。

根据仿真事件插入式排序的要求，选择AVL排序树作为全局事件队列的容器。值得注意的是，在离散事件仿真过程中，向队列中插入事件的时戳有这样的规律：插入事件的时戳都比较大，一般都是当前队列中最大的；而弹出的都是当前队列中时戳最小的事件。根据这个规律，对AVL排序树进行下述优化：

（1）保存AVL排序树中的最小时戳节点和最大时戳节点；

（2）在执行插入操作时先将插入时戳和当前最大时戳比较，若比最大时戳还大，则将其插为最大时戳节点的右子节点，否则就按一般情况从根节点开始比较插入；

（3）在执行弹出操作时，直接将最小时戳节点删除，并且找到最小时戳节点的父节点。如果父节点无右子树，即将父节点直接设置为最小时戳节点；否则将父节点右子树的最左子节点设置为新的最小时戳节点。

原有AVL排序树的插入和查找操作都要进行h（树的深度）次比较才能找到位置。而优化后，插入操作只需要一步比较操作就能找到插入位置；弹出操作不需比较，就能直接弹出最小时戳节点。只是在弹出操作后要确定下一个最小时戳节点，这需要几步比较操作。但是自平衡后的AVL排序树的最小时戳

节点和次小时戳节点的距离很近，至多两三步比较就可以完成操作，与 AVL 排序树的深度无关。

经过上述的分析，根据离散事件仿真中事件队列的特点，本节构造了基于优化 AVL 排序树的事件队列结构。这个队列主要由两部分组成：输入事件队列和中心事件队列。其中，输入事件队列是一个类 FIFO 队列，排序的操作很少；而全局事件队列的容器采用优化的 AVL 排序树。下面分别详细介绍如何在保守策略和乐观策略的时间同步的事件调度中应用这种单排序多层次事件队列。

5.3.3　保守时间同步策略下的基于优化 AVL 树的仿真事件调度算法

1. 保守时间同步策略下的基于优化 AVL 树的仿真事件队列

如图 5.5 所示，采用保守时间同步策略下的基于优化 AVL 树的仿真事件调度算法的混合仿真包含三种事件队列：输入事件队列、LP 模型事件槽和全局 AVL 排序树。其中，在 LP 层次有与其对应的模型事件信号灯槽（Atomic Event Signal Slots，AESS），在全局 AVL 排序树层次有与其对应的中心事件信号灯槽（Central Event Signal Slots，CESS）。下面分别介绍这些队列。

（1）输入事件队列：是每个原子模型的输入事件队列，根据前述离散事件仿真的特点，在模型输入端积累的事件队列是近似自然顺序的。因此排在队列顶端的事件根据模型事件信号灯弹入原子模型的输入端口。

（2）LP 模型事件槽：用于管理进程内所有原子模型时戳事件集合。事件槽的大小由进程内拥有的模型数量决定。每个模型对应一个事件槽，事件槽内始终压有一个对应模型的时戳最小事件。值得注意的是，LP 模型事件槽本身不具备排序能力，只按模型保存事件。

（3）全局 AVL 排序树：是整个仿真事件调度的中心，集合了所有原子模型的最小时戳事件。排序树的大小由仿真中所有模型的数量决定。排序树根据时戳信息进行排序，其最左子节点始终都是仿真中需要执行的下一个事件。

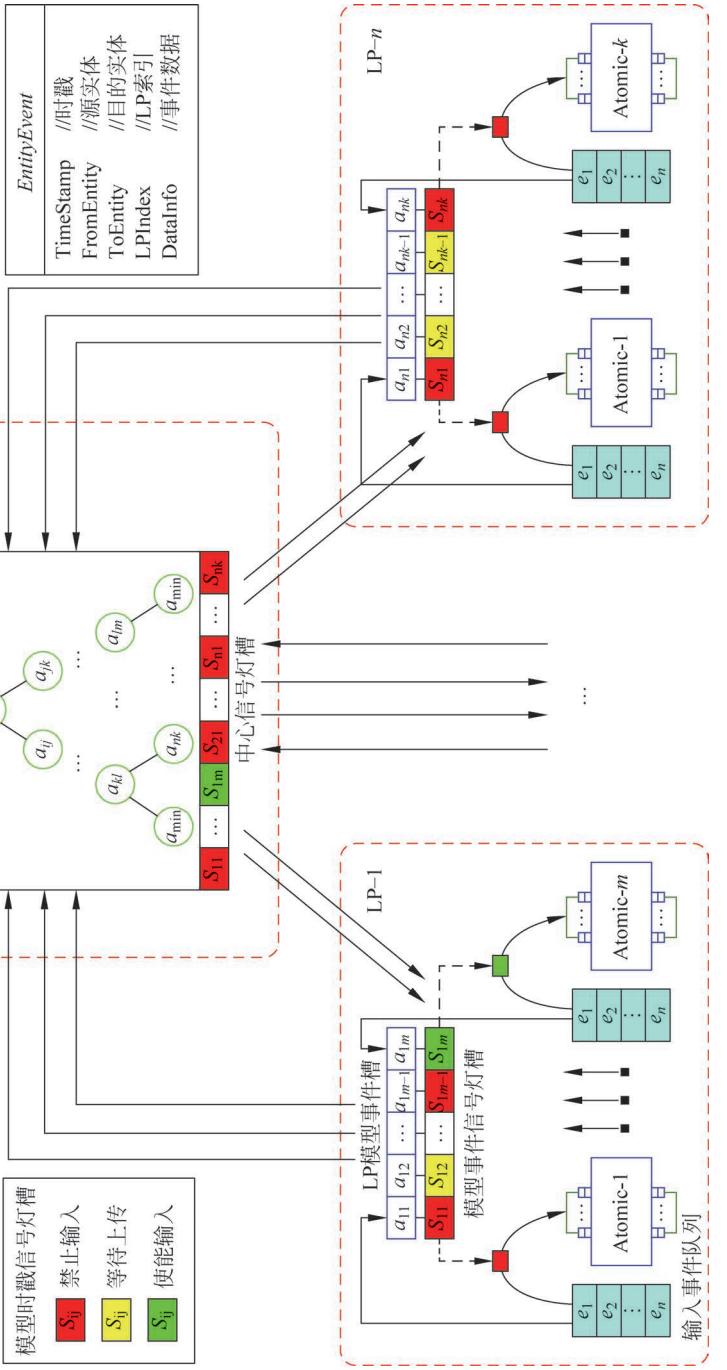

图 5.5　保守时间同步策略下的基于优化 AVL 树的仿真事件调度

(4) 模型事件信号灯槽(AESS)：与事件槽对应，保存从 CESS 传递来的模型事件信号灯。若信号灯为黄色，表示事件刚刚压入事件槽并向全局 AVL 排序树发送更新请求；若信号灯为红色，表示事件已经进入全局 AVL 排序树，但还未获得处理许可；若信号灯为绿色，表示事件已经获得处理许可，可以进入模型的输入端口，并向输入事件队列请求该实体下一个最小时戳事件。

(5) 中心事件信号灯槽(CESS)：与全局所有原子模型一一对应，CESS 只有红色和绿色两种状态，红色表示事件处于等待；绿色表示事件许可执行。

2. 保守时间同步策略下的仿真事件调度算法

调度算法建立在前述事件队列的基础上，分成本地 LP 和中心服务器两部分。在本地 LP 内部：

(1) 原子模型在收到输入事件后，将其保存在输入事件队列中；

(2) 输入事件队列将第一个事件弹入 LP 模型事件槽，并将对应模型的信号灯置为黄色；

(3) LP 在全局 AVL 树更新事件后将对应模型信号灯置为红色；

(4) LP 在某个模型的事件信号灯被置为绿色后将对应事件弹入模型的输入端口；

(5) 输入事件队列将下一个事件弹入 LP 模型事件槽，重复以上过程。

在中心服务器中：

(1) 中心服务器在收到更新的事件后，将其插入全局 AVL 排序树并更新中心事件信号灯槽，所有更新模型的信号灯均置为红色；

(2) 全局 AVL 排序树进行排序操作和自平衡操作；

(3) 全局 AVL 排序树弹出最小时戳的事件，中心服务器将该事件对应的信号灯置为绿色；

(4) 中心服务器将中心事件信号灯槽的最新状态刷新到模型事件信号灯槽。

保守时间同步策略下的仿真事件调度算法(CTESAlgorithm)的伪代码如

算法 5.1 所示。

算法 5.1　CTESAlgorithm

```
Input: InputEventQueue
Variables: LPS,Global
Output: GlobalEventScheduling
FIFO InputEventQueue[n][m][k]
Enumeration LPSignalSlot = { yellow,red,green }
Enumeration GlobalSignalSlot = { red,green }
Enumeration LPS[n][m]
Enumeration GlobalS[n * m]
AvlTree GlobalEventScheduling
for each LPS[i][j] in the LPS
    LPS[i][j] = { red }
end for
for each GlobalS[i] in the GlobalS
    GlobalS[i] = { red }
end for
While InputEventQueue != NULL do
    InputEventQueue[i][j][q].pop()
    LPS[i][j] = { yellow }
    LPS[i][j].RequestUpdate()
    GlobalEventScheduling.Push(LPS[i][j])
    LPS[i][j] = { red }
for each GlobalS[r] in the GlobalS
    GlobalS[r] = { red }
end for
GlobalEventScheduling.AutoSort()
GlobalEventScheduling.pop()
GlobalS[i] = { green }
LPS[i][j] = { green }
InputEventQueue[i][j][q]. execute()
LPS[i][j] = { red }
GlobalS[i] = { red }
End While
```

3. 时间复杂度分析

本书提出的多层次事件队列，分析其时间复杂度就要从研究每层次队列的时间复杂度出发。首先分析原子模型的输入事件队列。由于不需要排序，查找时只需要取出队列中的第一个元素即可，所以时间复杂度为 1；其次来看 LP 实体事件槽，它仅是一个存储事件的中间数据结构，不承担排序任务，所以此处时间复杂度可以近似认为是 0；最后是全局 AVL 排序树，向树插入一个节点的时间复杂度为 $O(\lg n)$，其中 n 为离散事件仿真中实体模型的数量。所以这种排序方法的时间复杂度是 $1+O(\lg n)$。由于实体数量 n 基本是固定的，所以时间复杂度是稳定的。与文献[135]中的执行两次排序过程的三层事件队列比较，本节提出的事件队列优势有两个方面：①应用了 AVL 排序树这种稳定的插入式排序数据结构，还针对离散事件仿真的特点进行了优化；②将其局部排序队列扩展为全局排序队列，将两次排序过程变为一次排序过程，降低了时间复杂度。文献[135]中排序算法的复杂度为 $1+O(\lg k)+O(\lg m)$，其中 $k\times m=n$。根据实验分析，n 在[0,100000]的范围内变化时，AVL 二叉排序树的插入排序时间几乎一致，所以应用本书的事件队列要节省 $O(\lg m)$的时间。

5.3.4　乐观时间同步策略下的基于优化 AVL 树的仿真事件调度算法

1. 乐观时间同步策略下的基于优化 AVL 树的仿真事件队列

如图 5.6 所示，乐观时间同步策略中每个 LP 的时间推进不受中心约束，由自身引擎分发事件和时间推进。因此，只需要构建局部 LP 的事件队列。保守时间同步策略中的三层事件队列结构变为两层，中心服务器只负责维护 GVT 的计算及公共管理控制事件的传输。排序队列建在 LP 层次，且 LP 之间相互独立。LP 内部的事件队列有输入事件队列、LP-AVL 排序树和 LP 模型事件信号灯槽(LP Event Signal Slots，LPESS)。

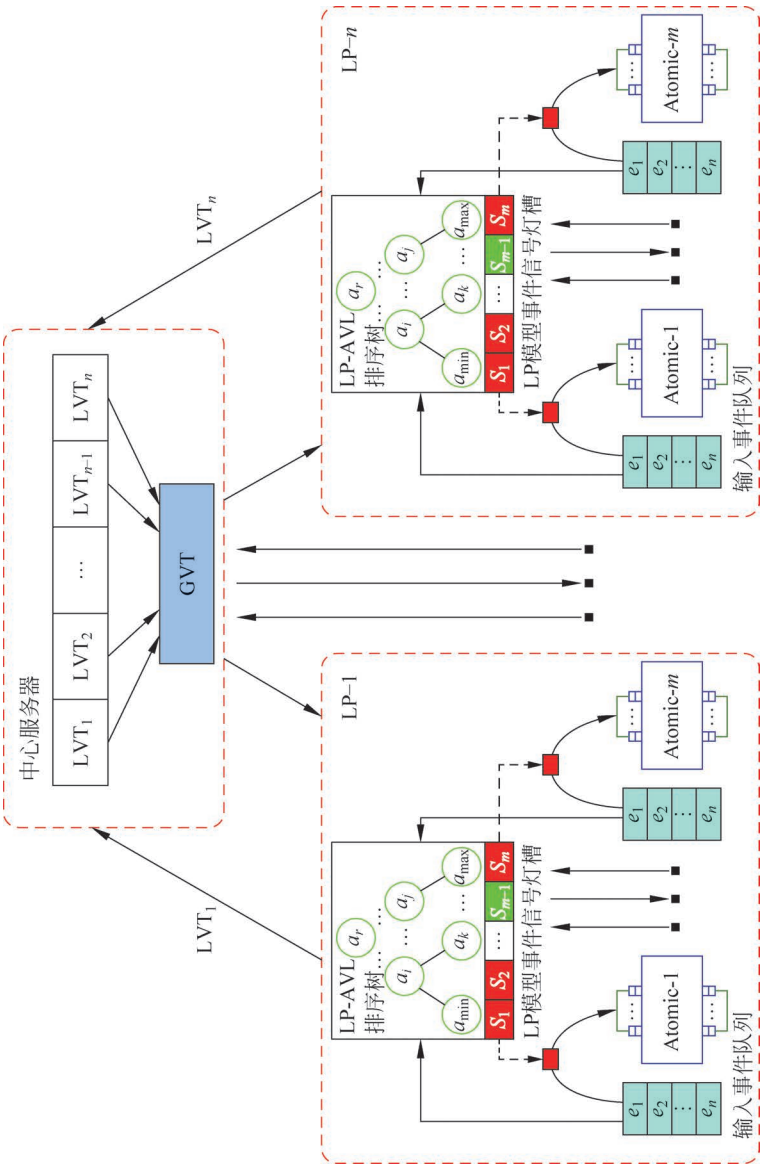

图 5.6　乐观时间同步策略下的基于优化 AVL 树的仿真事件调度

（1）输入事件队列：与保守时间同步中的输入事件队列一样，是一个 FIFO 队列。

（2）LP-AVL 排序树：在本地 LP 中的 AVL 排序树，直接插入输入事件队列中的首个事件进行排序，并弹出本地 LP 的最小事件。

（3）LP 模型事件信号灯槽 LPESS：因为 AVL 排序树已经转移到本地 LP，所以 AESS 也转移到本地成为 LPESS 负责控制每个事件的输入。由于不需要再向中心事件队列传输事件，因此 LPESS 的信号灯只有红色和绿色两种，含义不变。

2. 乐观时间同步策略下的仿真事件调度算法

介绍乐观时间同步策略下的仿真事件调度算法（OTESAlgorithm）前首先介绍以下一些基本概念。

（1）Straggler 事件：与本书 5.2.2 节中的概念一致，指的是时间戳小于实体的当前时间，违反因果关系的事件。当实体接收到 Straggler 事件时，就要回滚到前一个时间点。这个时间点就称作回滚时间（Rollback Time）。

（2）接收模型时戳列表（Receiving AtomicTimeStamp List，RATL）：每个原子模型需要挂接的时戳列表，保存以该模型为源发送事件的目的模型集合及其最近发送时间。若实体收到了 Straggler 事件或反事件，则根据事件的时戳，向每个最近发送时间大于该时戳的实体发送一个反事件。一旦列表中某个目的模型的接收时戳小于 LVT，将其从列表中删去。

（3）反事件（Anti-Event）：与 5.2.2 节中的概念一致，由于利用 RATL 代替了输出队列，因此此处反事件按实体发送，并不需要为每个要撤销的事件都发送一个反事件。反事件包括反事件标志、源模型 ID、目的模型 ID、反时戳等信息。利用反事件，目的实体模型完成回滚、撤销源模型输入事件以及发送反事件等操作。值得注意的是，反事件只存在于不同 LP 的模型之间，LP 内部的模型之间由于伪 Straggler 事件的作用不再需要反事件。

（4）伪 Straggler 事件：当一个 LP 收到一个 Straggler 事件或反事件时，LP 将该事件发给目的模型，同时向所有其他模型发送伪 Straggler 事件。这些模型根据该事件完成回滚等操作，伪 Straggler 事件只存在于 LP 内部。

由于事件队列的多层次性，使乐观时间同步策略下调度算法的实现也需要考虑两个层次：LP 层次和模型层次。特别是在 LP 层次，需要考虑某个模型的回滚对 LP 内其他模型的影响。因此，提出伪 Straggler 事件解决这个问题。下面首先给出 LP 收到 Straggler 和反事件的处理方式，如图 5.7 所示。本节注重采用事件队列结构对乐观算法带来的变化，不再冗述乐观算法本身的功能。

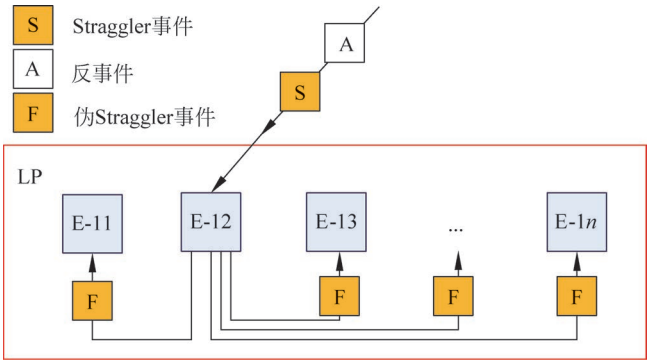

图 5.7　LP 接收 Straggler 事件或反事件的情况

（1）LP 收到 Straggler 事件：当某一模型收到 Straggler 事件时，说明该 LP 也收到了 Straggler 事件。该模型的回滚操作完成后，向 LP 内所有未收到 Straggler 事件的模型发送伪 Straggler 事件。伪 Straggler 事件包含模型 ID、回滚时戳等信息。当所有模型都回滚完毕后，LP 向模型事件时戳槽更新事件时戳并重新排序。

（2）LP 收到反事件：与收到 Straggler 事件类似，LP 收到反事件后也要群发伪 Straggler 事件。但是不同的是，LP 在收到反事件后立即产生伪 Straggler 事件，而不用等待反事件的目的模型产生回滚时戳。

在模型层次，由于采用了 RATL，接收事件的处理方式与传统乐观算法中时间弯曲的方法相比稍有变化，下面分别介绍模型接收 Straggler 事件、伪 Straggler 事件和反事件的情况。

（1）模型收到 Straggler 事件：首先，要将模型状态回滚到比 Straggler 事件时戳小的最近时间点（即为回滚时间），其次，整理输入事件队列，恢复到回滚时间之前的状态，同时也要将 Straggler 事件插入队列相应的位置上。再次，检

查接收实体时戳列表 RATL，向列表中所有时戳大于回滚时间的实体发送反消息。最后，将输入队列中最小时戳事件弹入本地 LP 的 AVL 排序树。

（2）模型收到伪 Straggler 事件：首先，将模型状态和事件队列都恢复到小于回滚时间的最近时间点上。其次，根据伪 Straggler 事件中的回滚时间和模型 ID，将输入队列中满足以下两个条件的事件都删除：①来自伪 Straggler 事件中源模型或本地模型；②时戳大于回滚时间。最后，根据回滚时间和 RATL 发送反事件。同样，也要将重整后的输入事件队列中的最小时戳弹入 LP-AVL 排序树。

（3）模型收到反事件：操作与收到伪 Straggler 事件类似，只是反事件来源于其他 LP。

乐观时间同步策略下的仿真事件调度算法（OTESAlgorithm）的伪代码如算法 5.2 所示。

算法 5.2　OTESAlgorithm

```
Input: StragglerEvent,AntiEvent,FakeStragglerEvent
Variables: RATL
Output: LPEventScheduling
1: FIFO InputEventQueue[n][m][k]
2: Enumeration LPSignalSlot = { red,green }
3: Enumeration LPS[n][m]
4: AvlTree LPEventScheduling[n]
5: for each LPS[i][j] in the LPS
6:    LPS[i][j] = { red }
7: end for
8: While InputEventQueue != NULL do
9:     InputEventQueue[i][j][q].pop()
10:   LPS[i][j].RequestUpdate()
11:   LPEventScheduling[i].Push(LPS[i][j])
12:   LPEventScheduling[i].AutoSort()
13:   LPEventScheduling[i].pop()
14:   LPS[i][j] = { green }
15:   InputEventQueue[i][j][q]. execute()
16:   LPS[i][j] = { red }
17: If LP[i].InputEventQueue.Push(StragglerEvent) == true
```

```
18:   ModelRollback()
19:   SendToOtherModel(FakeStragglerEvent)
20:   ModelRollback(allmodel)
21: End if
22: If LP[i].InputEventQueue.Push(AntiEvent) == true
23:   ModelRollback()
24:   SendToOtherModel(FakeStragglerEvent)
25:   ModelRollback(allmodel)
26: End if
27: If Model[i].InputEventQueue.Push(StragglerEvent) == true
28:   ModelRollback()
29:   Update()
30:   SendToOtherModel(StragglerEvent,RATL)
31: End if
32: If Model[i].InputEventQueue.Push(FakeStragglerEvent) == true
33:   ModelRollback(InputEventQueue)
34:   Update(InputEventQueue)
35: End if
36: If Model[i].InputEventQueue.Push(AntiEvent) == true
37:   ModelRollback(InputEventQueue)
38:   Update(InputEventQueue)
39: End if
40: End While
```

根据前述调度算法，图 5.8 给出一个基于 AVL 树的乐观时间同步中时间弯曲的例子。LP-1 中的模型 E-11 收到一个时戳为 4.5s 的 Straggler 事件。收到该事件后，E-11 首先恢复到时戳为 4.0s 的状态，包括模型状态的回滚和输入事件队列的恢复，并且将 Straggler 事件插入输入事件队列中。随后检查接收模型事件时戳列表，发现列表在 4.6s 分别向实体 E-12 和 E-21 各发送了一个事件，因此要发送反事件撤销这两个事件。由于 E-12 也在 LP-1 中，因此不用处理，只需要向 LP-2 的 E-21 发送一个回滚时间为 4.0s 的反事件。同时，LP-1 根据回滚时间向模型 E-12 发送伪 Straggler 事件。E-12 将状态回滚到 4.0s，且将事件队列中来自 E-11 的事件 $e_1(4.6)$ 删除。检查接收模型事件时戳列表，发现在 3.8s 向 E-22 发送了一个事件，由于 3.8s 小于回滚时间，因此不用处理。

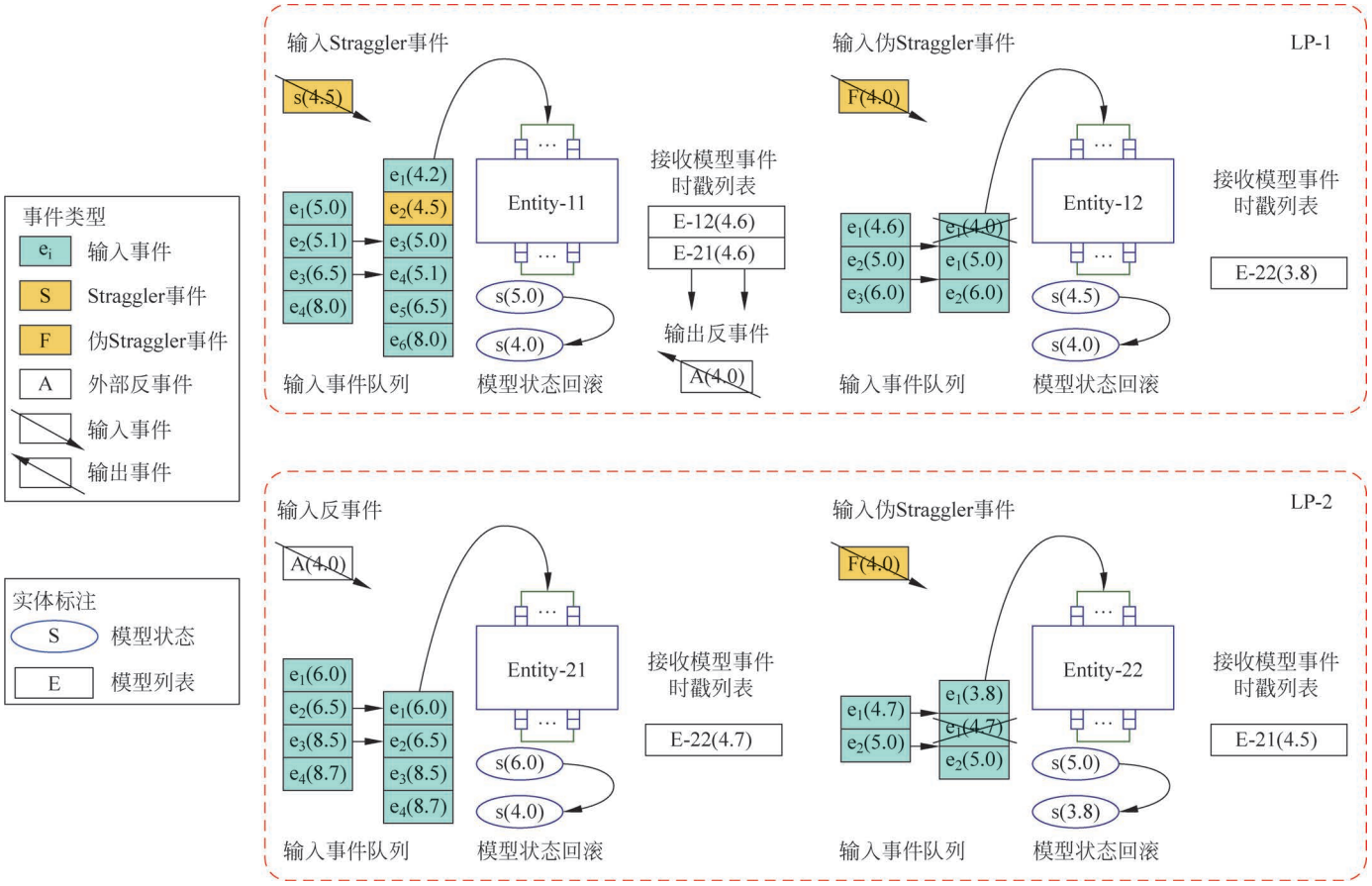

图 5.8　乐观时间同步策略下的基于优化 AVL 树的时间弯曲机制

LP-2 收到了回滚时间为 4.0s 的反事件后首先将反事件传递给反事件的目的模型 E-21；再向模型 E-22 发送时戳为 4.0s 的伪 Straggler 事件。模型 E-21 收到反事件后，先将自身状态回滚到 4.0s，随后在输入事件队列中删除来自本地 E-22 的事件 $e_1(4.5)$和 E-11 的事件 $e_2(4.6)$。检查接收模型事件时戳列表，只有向 LP 内 E-22 发送的事件的记录，所以不用处理。而模型 E-22 收到伪 Straggler 事件后首先将状态回滚到 3.8s，随后将来自本地 LP 的事件 $e_1(4.7)$删除。检查接收模型事件时戳列表，只有向 LP 内 E-21 发送的事件的记录，所以不用处理。至此算法完成了接收到一个 Straggler 事件的时间弯曲过程。

3. 调度算法的优势

乐观时间同步策略下的基于优化 AVL 树的事件调度算法以模型的输入事件队列为核心，与以往的乐观时间同步算法相比有以下几个优点：

(1) 用接收模型时间时戳列表 RATL 代替输出事件队列(Output Event Queue)，由于 RATL 只针对每个目的模型保存一个时戳值，而不会根据事件的增加而累加，相对于以前输出事件队列详细记录所有输出事件节省了许多空间，降低了空间复杂度；

(2) 采用伪 Straggler 事件代替了在本地进行的反事件传递，一方面通知了所有模型将状态恢复到回滚时间，另一方面也将事件队列中来自本地模型和 Straggler 事件或反事件源模型的事件都删除。只需要发送一次伪 Straggler 事件就能够保证所有模型状态的回滚。

(3) 对于不同 LP 上的模型，只需要发送一个反事件，就足以删除目的模型输入事件队列中的对应输入事件。这比利用输出事件队列要将所有的反事件发送给目的模型大大节省了网络传输开销，降低了模型之间和 LP 之间的耦合度。

5.3.5 基于排队论的仿真事件调度分析

在分布式仿真系统中，仿真事件队列性能也是影响仿真效率的重要因素，其指标主要有队列的平均队长、每个事件的平均等待时间和系统能够容纳的队

列个数。本节基于排队论方法，对多层仿真事件队列处理机制中 LP 的事件输入队列和中央服务器中的全局队列进行建模，分析传统事件队列和多层事件队列的性能差别，以及分别采取保守和乐观事件调度策略时系统的性能差别。

1. 传统的仿真事件调度

在传统的仿真事件队列中，原子模型队列将事件传输至 LP 排序队列，然后通过事件调度算法，调度至中央服务器进行排序和处理，由于原子模型队列为 FIFO 队列且不对事件进行处理，因此不需要计算其队列性能，考虑 LP 排序队列即可。

将 LP 事件输入队列看作到达率为 λ_{ATO} 的 FIFO 队列，由于 FIFO 队列不对事件排序，仅有 LP 队列排序，可以设每个 LP 有 N 个事件输入队列，那么每个 LP 队列的到达率为 $\lambda_{\mathrm{ATO}}=N\lambda_{\mathrm{ATO}}$，中央服务器的事件处理服务率为 μ。假设当前系统有 k 个 LP，对于每个事件检查时戳的时间为 c，那么对 LP 事件输入队列的服务时间为 $k/u+ck^2$，对于单个 LP 事件输入队列的服务率为

$$\mu'=\frac{\mu}{k(1+k\mu c)} \tag{5-3}$$

因此，可得单个 LP 事件输入队列的服务强度为

$$\rho'=\frac{\lambda}{\mu'}=\frac{k\lambda(1+k\mu c)}{\mu} \tag{5-4}$$

为了保证事件处理引擎能够及时处理所有的事件，确保仿真能够正常运行，则需要假设 $\rho'\leqslant 1$，则有系统中的 LP 数不应超过：

$$k\leqslant \min\left\{\frac{\lambda}{u},\frac{\sqrt{\lambda^2+4\mu^2c\lambda}-\lambda}{2\mu c\lambda}\right\} \tag{5-5}$$

若给定 LP 数量，则每个事件队列检查时戳的时间不应超过：

$$c\leqslant\frac{\mu-\lambda k}{\lambda\mu k^2},\quad k\leqslant\frac{\lambda}{\mu} \tag{5-6}$$

在此情况下，每个 LP 的事件的队列平均长度 L_q 为

$$L_q=\frac{(\rho')^2}{1-\rho'} \tag{5-7}$$

由于每个事件到达时，队列需要对事件的时戳进行排序，该动作的复杂度

为 $O(n)$，那么每个事件的平均等待时间 W_q 为

$$W_q = \frac{\lambda}{\mu'(\mu' - \lambda)} + cN \tag{5-8}$$

2. 保守时间同步策略下的仿真事件调度

采用保守时间调度策略时，对于多层次队列来说，在每个 LP 事件队列部分，调度机构需要轮训各个 LP，找出其中时戳最小的事件，则可知服务率为 $\mu_{co} = \frac{1}{ck^2}$，那么服务强度为

$$\rho_{co} = \lambda c k^2 \tag{5-9}$$

同理可得，LP 的个数应为

$$k_{co} \leqslant \sqrt{\frac{1}{\lambda c}} \tag{5-10}$$

在 LP 队列阶段，由于 LP 队列仅排序 N 个事件，因此每个 LP 队列的平均队长和每个事件的平均等待时间分别为

$$L_{co} = \frac{(\rho_{co})^2}{1 - \rho_{co}}, \quad W_{co} = \frac{\lambda}{\mu_{co}(\mu_{co} - \lambda)} + cN \tag{5-11}$$

在全局事件队列中，队列中的事件到达率为

$$\lambda_{AVL} = \frac{1}{ck^2} \tag{5-12}$$

则同理可得服务强度、平均队长分别为

$$\rho_{AVL} = \frac{1}{c\mu k^2}, \quad L_{AVL} = \frac{(\rho_{AVL})^2}{1 - \rho_{AVL}} \tag{5-13}$$

由于采用 AVL 树进行排队，因此可得到每个事件的平均等待时间为

$$W_{AVL} = \frac{1}{\mu(ck^2\mu - 1)} + c \lg \left[\frac{(\rho_{AVL})^2}{1 - \rho_{AVL}} \right] \tag{5-14}$$

那么可知在本书设计的事件队列结构下，总的事件平均等待时间为

$$W_{total} = \frac{1}{\mu(ck^2\mu - 1)} + \frac{\lambda}{\mu_{co}(\mu_{co} - \lambda)} + c \lg \left[\frac{(\rho_{AVL})^2}{1 - \rho_{AVL}} \right] + cN \tag{5-15}$$

3. 乐观时间同步策略下的仿真事件调度

采用乐观时间同步的事件调度时，每个 LP 在本地进行事件排序和处理，并不向中央服务器发送事件进行排序。因此，令每个 LP 的事件处理引擎服务率为 $\mu_{op} \leqslant \mu$，那么可得服务强度为

$$\rho_{op} = \frac{\lambda}{\mu_{op}} \tag{5-16}$$

进而可得

$$L_{op} = \frac{(\rho_{op})^2}{1-\rho_{op}}, \quad W_{op} = \frac{\lambda}{\mu_{op}(\mu_{op}-\lambda)} + c\lg(N) \tag{5-17}$$

4. 结果分析

1）系统支持的 LP 个数

根据式(5-15)和式(5-10)，可以得出

$$\sqrt{\frac{1}{\lambda c}} > \min\left\{\frac{\lambda}{u}, \frac{\sqrt{\lambda^2 + 4\mu^2 c\lambda} - \lambda}{2\mu c\lambda}\right\} \tag{5-18}$$

证明：

$$\begin{aligned}\sqrt{\frac{1}{\lambda c}} &= \sqrt{\frac{(\lambda^2 + 4\mu^2 c\lambda - \lambda^2)}{4\mu^2\lambda^2 c^2}} \\ &= \sqrt{\frac{(\lambda^2 + 4\mu^2 c\lambda)}{4\mu^2\lambda^2 c^2} - \frac{\lambda^2}{4\mu^2\lambda^2 c^2}} > \frac{\sqrt{\lambda^2 + 4\mu^2 c\lambda}}{2\mu c\lambda} - \frac{\lambda}{2\mu c\lambda}\end{aligned} \tag{5-19}$$

由式(5-18)可知，采用本书设计的事件队列结构，在采用保守的事件调度策略时，相较于传统的事件队列结构，同样服务率的中央事件处理引擎，可以支持更多的 LP 个数。而采用乐观的事件调度策略时，则对于系统支持的 LP 个数没有限制。因此，采用本书设计的多层事件队列结构，可以有效增强分布式仿真系统的可扩展性。

2）LP 事件队列的平均队长

通过对比三种队列结构和调度算法的服务强度，可以得出

$$\rho' = \frac{k\lambda(1+k\mu c)}{\mu} = \lambda k^2 c + \frac{k\lambda}{\mu} > \rho_{co} = \lambda c k^2 \tag{5-20}$$

进而可以得出

$$L_q > L_{co} \tag{5-21}$$

因此，可以得到采用多层队列能够有效减少 LP 中的事件队列长度，减少 LP 中的缓冲区大小，从而有效节省成本。但是，这里同样需要认识到，在采用保守事件调度时，LP 中事件队列的减少是以中央服务器中队列的增长为代价的，其队列长度可以表示为

$$L_{AVL} = \frac{1}{(\mu k^2 c - 0.5)^2} - \frac{1}{4} \tag{5-22}$$

可见中央服务器中的队列长度随着 c 和 k 的增加而减少，因为当检查时间和 LP 数量增多时，事件被发往中央服务器的频率将下降，而每个 LP 中的队列长度则会增长。此外，根据乐观事件调度策略的原理可知，当 $\mu_{op} \geqslant \frac{1}{k^2 c}$ 时，乐观事件调度相比于保守事件调度策略具有更短的队列。因此可知，当 LP 的处理能力更强时，采用乐观事件调度策略，仿真运行更有效率。

3）每个事件的平均等待时间

通过对比式(5-8)、式(5-15)、式(5-17)，可知

$$W_q > W_{co} > W_{op} \tag{5-23}$$

在传统事件队列中，由于采用复杂度为 $O(n)$ 的排序算法，导致每个事件的平均等待时间稍有增长。而采用乐观事件调度策略时，由于每个事件可以在本地排序和处理，因此使得每个事件的平均等待时间具有明显的改善，进而可以明显提升仿真运行效率；但是与此同时，仍需考虑当事件执行出现错误时，产生的回滚时间。通过以上分析，可知采用本书设计的多层事件队列和调度算法，能够在一定程度上提升仿真系统的性能。

5.3.6 测试用例

1. 用例描述

本书用仿真实验验证这种基于 AVL 排序树的事件调度算法对仿真性能带来的提升。仿真实验分为三部分，前两个部分是在保守时间同步中测试不同排

序方法或不同事件队列结构对仿真性能的影响；第三个部分则测试不同排序方法在乐观时间同步中对仿真性能的影响。

第一个实验是测试在单排序多层次事件队列结构上采用不同排序方法时的性能指标，分别对优先级队列、AVL 排序树以及本书提出的优化的 AVL 排序树方法的仿真结果进行对比分析。仿真场景采用 2 个 LP，事件规模从 0 到 10000 变化，每间隔 50 个事件采样一次，结果如图 5.9 所示。横坐标是事件数量，纵坐标是仿真消耗时间，单位为秒。

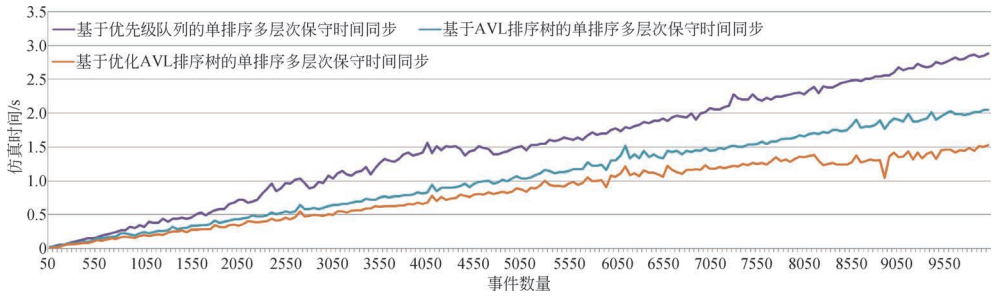

图 5.9　保守时间同步策略下的单排序多层式事件队列的性能比较

第二个实验是测试优化的 AVL 排序树在不同事件队列结构时的性能指标，分别对全局唯一事件队列、本书采用的单排序多层次事件队列、双排序多层次事件队列的仿真结果进行了对比分析。仿真场景采用了 2 个 LP，事件的数量在 10000～40000 变化。每隔 500 个事件采样一次，实验结果如图 5.10 所示。横坐标表示仿真事件个数，纵坐标是仿真消耗时间，单位为秒。

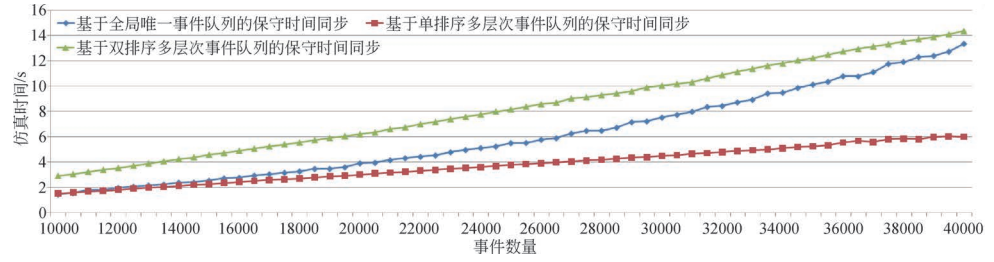

图 5.10　保守时间同步策略下采用优化 AVL 排序树的事件队列结构的性能比较

第三个实验是测试在乐观时间同步中采用不同的排序算法时的性能指标，分别对优先级队列、AVL 排序树、优化的 AVL 排序树的仿真结果进行对比分

析。仿真事件数目固定为 10000 个。为了体现乐观时间同步的性能随 LP 变化的情况，在 LP 数目变化的情况下采样性能数据。实验结果如图 5.11 所示。横坐标是 LP 的数量，纵坐标是仿真消耗的时间，单位为秒。

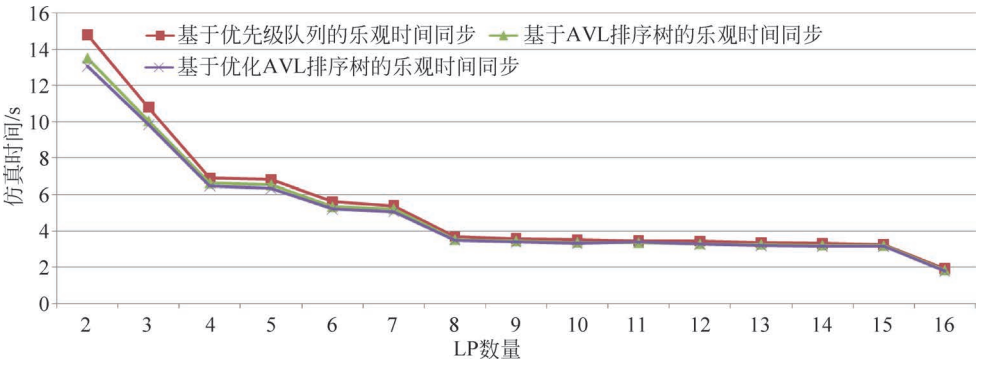

图 5.11　乐观时间同步策略下的三种排序容器的性能比较

2. 结果分析

由图 5.9 可以看出，随着事件数量的增加，采用优化 AVL 排序树的调度算法性能最优，AVL 排序树次之，而优先级队列的性能最差。这是由于优化 AVL 排序树是在 AVL 排序树的基础上针对离散事件队列的特点进行优化。优先级队列是基于堆排序的一种队列，平均时间复杂度为 $O(n\lg n)$，因此在动态变化的事件队列中不能提供高性能。通过图 5.9 中曲线可以看出，在趋近 10000 个事件时，优化 AVL 排序树的执行时间是优先级队列的 1/2，比 AVL 排序树快 30%。

由图 5.10 中可以看出，本书提出的单排序多层次事件队列的性能最优，全局唯一事件队列其次，而双排序多层次事件队列的性能最差。根据前述的时间复杂度分析，全局唯一事件队列的事件复杂度为 $O(\lg n)$，大于单排序多层次事件结构 $1+O(\lg m)$，其中 m 是实体数量，n 是全局唯一事件队列的事件数量。由实验可以看出单排序多层次事件对结构对时间的消耗随时间以一个比较小的斜率增长，而全局唯一事件队列则随时间快速增长，呈现一个不稳定的增长状态，在 40000 个事件时已经接近了双排序多层次事件队列。这和全局唯一事件队列的动态性是相吻合的。双排序多层次的复杂度为 $1+O(\lg m)+O(\lg k)$，其

中 m 为实体数量，k 为 LP 数量。这里的 1 就是指从实体事件队列中取出第一个事件，几乎可以忽略。而在 AVL 排序树中，$O(\lg m)$ 和 $O(\lg k)$ 基本相等，所以双排序多层次事件队列的时间消耗约等于单排序多层次事件队列的 2 倍，这也在图 5.10 中得到了验证。

由图 5.11 中可以看出，在乐观情况下不管是哪种排序队列，仿真性能随着 LP 数量的增加呈类线性提高。而在这三种排序方法中，乐观时间同步下的优化 AVL 排序树性能最优，AVL 排序树次之，而优先级队列的性能最差。这和保守时间同步中的情况一致。需要说明的是，随着 LP 数量的增加，性能提升的幅度越来越小。在两个 LP 的情况下，AVL 排序树比优先级队列提升了 8%，而优化 AVL 排序树又在 AVL 排序树的基础上提升了 3%。但是到 16 个 LP 时，提升则分别只有 2%和 0.8%。这是由于 LP 增多，每个 LP 内部需要排序事件数量减小，而仿真时间是根据完成仿真消耗时间最长的 LP 确定的。每个 LP 中需要排序的工作量越来越小，因此性能的提升也越有限。

5.4　小结

时间管理方法对面向装备智能化保障体系的分布式混合仿真至关重要。仿真时间管理是仿真系统中各功能模块的基础，也是仿真正确运行的重要保障，具有非常重要的研究意义。本章深入研究了分布式混合仿真的时间同步策略，在分析保守和乐观的时间同步策略的基础上，提出混合的时间同步策略，减小了同步计算量，提高了仿真的并发能力。针对仿真事件调度效率低下的问题，提出一种基于优化自平衡二叉排序树的分布式混合仿真事件调度优化方法，并进行了案例验证。实验结果表明，该方法一定程度上优化了仿真运行时间。