Real-Time 并行编程：现状与未决问题

Luis Miguel Pinho

波尔图理工学院工程学院（*ISEP*）与 *INESC TEC*

葡萄牙波尔图

摘要

实时系统应用程序通常由一组具有时序相关属性的并发活动组成。开发这些应用程序需要采用编程范式，以有效处理并发活动和时间约束的规范，并控制它们在特定平台上的执行。追求高性能的主流趋势以及细粒度并行执行的应用，使这项任务更具挑战性。

本文概述了实时并行应用开发的最新进展与挑战，重点关注当前的两个研究方向：一个来自高性能计算领域（基于 *OpenMP*），另一个来自实时系统与关键系统领域（基于 *Ada*）。本文回顾了这两种方法各自支持的特性，指出了尚待解决的问题和当前的研究方向。

关键词：实时、并行计算、编程、OpenMP、Ada

1. 引言

实时系统除了满足功能需求外，还需要在特定时间间隔（截止期限）内提供结果；这些时间间隔由应用程序需求决定[1]。这类系统是一个重要且颇具挑战性的领域，应用范围广泛，从工业自动化到物联网，都是“信息物理系统”的基石：在大多数此类应用中，计算系统会与外部环境交互，感知并执行相应操作[2]。在这种情况下，计算系统结果的时序正确性对于物理环境的正常运行至关重要。

截止期限的关键性基本分为硬实时和软实时两类[1]。对于硬实时截止时间，若未能在规定间隔内提供结果，就意味着计算系统发生故障；而对于软实时，错过截止时间可能导致性能下降，但行为仍可接受。大多数信息物理系统都有硬实时要求，其中某些截止期限必须严格满足。在大多数情况下（例如航空电子和汽车领域），信息物理系统用于关键应用程序，因此错过截止时间可能造成严重后果。

[1] 虽然还存在其他类别，但在大多数情况下，区分硬实时和软实时就足够了。

实时系统的典型特征是事件与物理相关活动之间的相互作用，这些事件和活动会在这些时间约束下按顺序或并发发生。[2]活动可以是周期性的，也可以是非周期性的（即事件驱动的）。周期性活动会在固定时间间隔后定期重复。非周期性活动又分为偶发活动和非周期活动：前者可以推导出重复的最高频率，后者的释放频率则没有上限[3]。

为了分析应用程序的截止期限能否得到满足，面临的一项挑战是确定应用程序各项活动在系统所用特定平台上的响应时间（即触发计算活动的触发事件与结果或操作输出之间的时间间隔）。目标是获得最坏情况值，即所有应用程序执行过程的安全上界。尽管这项工作最终可能并不复杂，但并发活动之间的干扰以及硬件架构日益增加的复杂性，带来了重大挑战。

因此，在计算系统中执行这些活动所采用的方法至关重要[3], 。人们提出了许多实时计算映射与调度方法，以保证结果能在有界且已知的时间内（原则上，在截止期限之前）提供。作为计算映射与调度的补充，实时调度理论致力于设计应用程序、调度器和平台模型，并开发工具与技术，使人们能够预测、建模和分析整个系统的时间行为。这些工具与技术可分为两大类，且彼此有别：时间分析和可调度性分析。

在时序分析中，目标是计算单独执行某项操作所需时间的紧界（或概率分布曲线），例如确定活动 [4] 的最坏执行时间（WCET）。在可调度性分析中，目标是在设计时或运行时通过分析检查系统的所有时序要求能否得到满足 [3], ，同时考虑最坏情况执行时间以及活动之间的相互作用和干扰。

现代信息物理系统的处理需求不断提高，这对实时系统的设计方法提出了挑战 [5]。当前，通过制造包含多个处理器核心的芯片来提升处理能力的趋势，使得利用细粒度并行性执行系统固有的并发活动日益普及。这种低成本、易获取的计算能力已使并行编程成为软件开发人员需要关注的问题，因为顺序编程模型难以适应此类多核系统的扩展需求 [6]。此外，时序要求带来的约束以及系统与物理世界之间的紧密交互，使这一问题更加棘手 [7]。

细粒度并行性的引入也影响了实时系统的发展，其中一个方面是为程序员提供的、用于

[2] 在实时系统领域，“任务”一词常用于指代此类计算活动。然而，在计算领域，“任务”一词的含义相当宽泛，其他相关用法包括：- 在 OpenMP 中，任务指由线程执行的可执行代码和数据环境的特定实例。- 在 Ada 中，任务是用于并发编程的语言实体（可用于实现实时任务）。- 在操作系统中，任务可能代表用于描述线程的内核结构。

开发软件应用程序。实时编程范式需要提供相应的能力，以表达和控制活动的并发执行与并行执行，从而实现时序安全的系统开发。

本文介绍了并行编程融入实时系统的最新进展。重点是经典实时编程范式 [8]，，也称为异步范式 [9] 或调度模型 [10]。

本文结构如下。下一节 概述实时系统和并行模型相关主题，并介绍并行性如何影响实时系统的开发。随后，第 3 节简要介绍实时编程方法，并以 POSIX 规范和 Ada 编程语言中的示例加以说明。

第 4 节阐述了并行实时系统编程模型的要求，并提出两种不同的方法来支持并行实时编程，分别在第 5 节和第 6 节讨论。第 5 节重点介绍 OpenMP 规范，尤其是 OpenMP 任务模型；第 6 节则着眼于即将修订的 Ada 语言标准。这两个示例从不同角度探讨了并行性与实时性的融合问题：OpenMP 全面支持并行编程，目前的工作是探索如何将其用于实时系统（可能借助 POSIX 支持）；另一方面，Ada 已为实时系统编程提供了广泛支持，而新版本中正在标准化的功能将加入并行性支持（可能构建于 OpenMP 运行时之上）。

2. Real 实时与并行模型

实时系统的基本计算模型是：一个应用程序由一组在同一硬件平台（单核或多核）上执行的周期性活动构成，这些活动竞争相同的资源，并采用明确的方法来管理其在平台（单处理器或多核）上的执行。调度算法涵盖了从静态分配活动优先级（固定优先级调度），到在执行期间计算出处理器上优先级最高的活动（最著名的例子是最早截止时间优先调度）。此外，现有方法既考虑活动的抢占式调度，也考虑非抢占式调度（前者中，如果优先级更高的活动变为就绪状态并可执行，就会抢占正在执行的活动），还考虑更具协作性的调度方式，例如基于服务器的调度或轮询调度（活动按预定义的预算共享处理器时间）。图 1 给出了一个示例：3 个活动在单个处理器上执行，采用固定优先级抢占式调度[3] 。

[3] 活动的垂直排列通常用来表示活动的优先级顺序（越靠上表示优先级越高）。

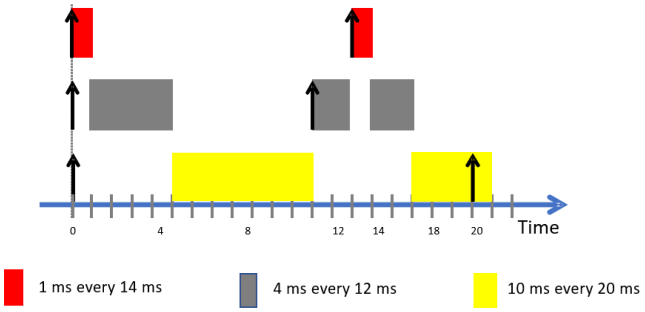


图 1. 固定优先级调度。

细粒度并行编程模型使程序员能够专注于展现可用的并行性，并在并发活动中表达这种并行性，而并行执行则由运行时系统动态管理。这意味着，活动不再指定一段代码的执行顺序，而是指定一个计算图，其中包含一组相互通信的代码段，这些代码段可以分叉或汇合通信路径。

在实时系统领域，为了更详细地描述并行活动结构，人们提出了以下模型（图 2）：

- 分叉/汇合模型 [11], 其中，每个活动都被划分为顺序执行段和并行段。并行段前后都必须有一个顺序执行段，且线程数不得超过平台上的处理器数。

- 同步并行模型 [12], 该模型通过将活动视为由不同执行段组成（这些执行段可包含任意数量的并行线程，甚至多于核心数），对分叉/汇合模型进行了泛化。

- 并行DAG模型 [13], 进一步推广该模型，将每个活动视为一个有向无环图（DAG），其中每个节点表示一个顺序代码执行段，每条有向边表示两个执行段之间的优先约束。

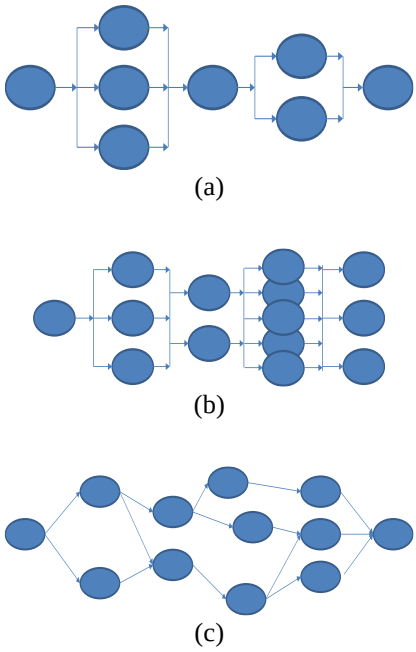


图 2。分叉/汇合（a）、同步（b）和并行有向无环图（c）模型。

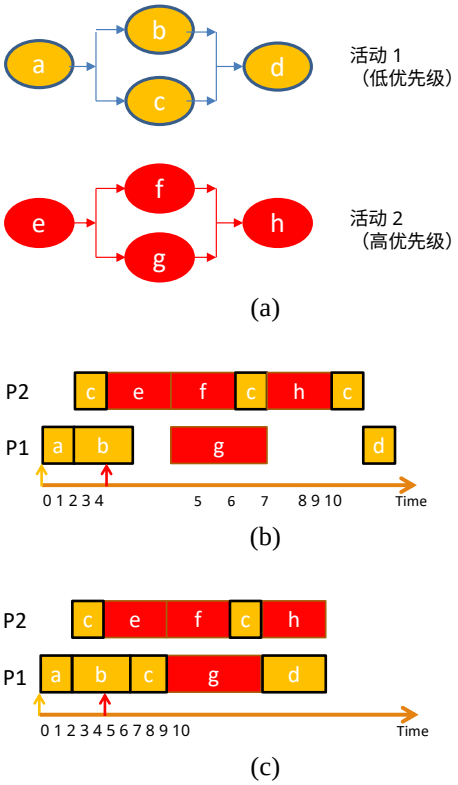
请注意，活动的执行时间现在是一组执行段的执行时间，而响应时间现在定义为该图的端到端响应时间。每种模型都提供了不同的方法，用于评估端到端响应时间是否允许执行在最坏情况下满足应用程序的时序要求（截止时间）。

2.1 并行计算的映射与调度

为了满足并行应用的时序要求，必须为系统提供适当的手段，将计算活动映射到底层操作系统线程（映射），并动态调度这些线程，以同时实现可预测性和高性能（调度）。

由于映射到同一核心上的多个活动之间存在干扰、不同核心访问共享平台资源（例如内存）时会相互干扰，以及活动之间共享数据，并行平台中并行代码段到核心的映射会显著影响应用程序的时序行为。

图3 (a) 展示了一个简单示例：两个活动在配备两台处理器的平台上执行（采用固定优先级抢占式调度）。一种简单的映射方式是将活动 1 映射到处理器 1，但将并行段 ‘c’ 映射到处理器 2（以实现并行性）；将活动 2 映射到处理器 2，但并行段 ‘g’ 除外。



Fi图3。映射示例：(a) 活动；(b) 无迁移的映射；(c) 有迁移的映射 n.

然而，如图3 (b) 所示，如果活动 2 在时刻 2 释放，就会出现两个时隙：此时处理器（P2）无法执行任务，尽管仍有待处理活动。在时刻 3，P2 无法执行执行段 ‘d’，因为 ‘c’ 尚未完成；也无法执行 ‘g’，因为 ‘e’ 也尚未完成[4]。在时刻 7，‘h’ 在 P1 上抢占了 ‘c’，而 P2 仍无法执行 ‘d’。如果允许 ‘c’ 在处理器之间迁移（图3 (c)），就能消除空闲时隙，因为 P2 可以在时刻 3 和 7 执行 ‘c’，从而缩短总体响应时间。

如图3所示，映射可以是动态的，即在执行过程中决定（c）；也可以是静态的，即在执行前“强制”确定（b）。动态映射可以带来更好的平均性能，而静态映射则可能提供更好的可预测性（最优与最差情况之间的抖动有所降低）[14]。如果动态映射还支持迁移，如图3（c）所示，也可能产生开销，因为在处理器之间迁移执行任务并非没有计算成本。

在并行平台上调度一组非并行活动，一直是实时系统领域广泛研究的课题 [15]。对于多处理器上的硬实时应用，分区调度占据主导地位，即将活动子集分配给特定处理器。这样便可以采用性能良好、易于实现且历史悠久的单处理器调度算法。此外，除与共享资源的交互外，这种方式还能在很大程度上为不同处理器上的活动提供时间隔离和功能隔离。

[4] 当考虑多个处理器时，非并行活动也会遇到这个问题。然而，并行性会使问题大幅加剧，同时也会使可考虑的不同映射与调度方案数量激增。

然而，如图3 (b)所示，分区调度常因碎片化而浪费处理能力。相反，完全迁移式调度允许活动在处理器之间迁移，甚至可在执行过程中途迁移，因此能够克服此类利用率问题，但也会带来实现和同步开销、破坏时间隔离，并增加分析难度。人们对混合“半-分区式”方案的兴趣日益浓厚；这类方案仅允许部分活动迁移，且迁移方式受到严格控制，旨在兼取分区调度和迁移式调度的优点。

对于采用分叉/汇合或同步并行模型构建的实时应用程序，并行代码段可以映射到一组操作系统线程；这些线程可以作为独立实体映射到各个处理器上，彼此之间没有优先约束，也可以进行全局调度，此时只有在所有此类并行段完成后，后续执行段才允许继续（由单个线程执行）[11]。扩展[12][13] 允许不同的并行阶段（线程数各不相同）直接交替进行，中间不经过顺序阶段。在这种情况下，可以将并行任务以不同方式映射到线程，即使采用静态映射方法也不例外，以提高系统利用率，同时保持可预测性 [16][17]。还可以采用更动态的方法 [18]，，让实时活动也使用工作窃取[19] 来实现负载均衡；此时采用动态映射，实时调度器可在核心空闲时允许代码块在线程之间迁移。

2.2 并行实时应用的时序与可调度性分析

在并行处理器中，由于核心之间共享的资源数量众多，确定最坏情况执行时间上界的问题更为复杂 [20]。在这类平台上，活动可能在不同的处理器核心上并发执行，但它们对内存的访问可能会与其他核心上运行的活动所发出的访问发生争用。在复杂架构中，核心因等待内存或总线请求得到处理而停顿的时间，是程序总执行时间中的重要组成部分；尤其是在代码块可能被动态映射到不同核心的情况下，这段时间可能会有很大变化[21]。

并行模型的可调度性分析目前是实时系统领域的一个重要研究主题（示例见[11][12][13][22][23]）。然而，尽管人们已开展大量研究，探索如何在并行平台上实现可预测行为的技术，时序与可调度性分析的前沿研究距离提供成熟的解决方案仍相去甚远。问题在于，如何刻画并行系统中各种硬件组件之间可能出现的多种干扰和相互依赖关系。当多个活动在同一处理器上同时执行，或分别在不同处理器上执行时，它们访问共享资源（如共享内存、通信网络或加速设备）可能会遭遇严重争用。时序与可调度性分析必须考虑并纳入这种争用，这可能导致极为悲观的响应时间界限。事实上，考虑极端情况的最坏情况场景（例如所有核心同时访问同一资源）可能会大幅增加执行时间和响应时间，或者在实践中难以处理 [14]。随着核心数量增加，这种膨胀会呈指数级增长，因此必须为这些罕见情况预留大量处理能力，从而造成严重的利用率不足，

系统。因此，为了减少这种悲观性，实用方法往往会对并行计算进行静态映射，并采用分区调度系统，从而提高整体保证利用率 [16][17]。

并行计算的映射与调度，以及相关的时序和可调度性分析，是一个内容十分丰富的研究课题，但与本文的主题并不直接相关。相反，本文关注同样重要的一个方面：为程序员提供相应的能力，使其能够表达和控制活动的并发与并行执行，从而开发出在时序方面安全的系统。

3. 实时系统编程

为了满足实时系统的特定需求，编程模型和范式通常会提供一组抽象，旨在体现实时系统的调度模型 [8][9][24]:

- 时间概念：在实时系统中，时间基准的表示方式和精度至关重要。

- 实时活动的描述：编程模型应尽可能支持描述重复发生的、由时间驱动的（周期性）活动，或由事件驱动的（偶发或非周期）活动。

- 活动属性的描述：在实时系统领域，为了确保受控系统的活动在截止期限前完成，调度方法会依据优先级、截止时间、执行时间等属性，因此编程模型应支持这些属性。

- 通信与同步：提供机制，使活动能够共享数据，并在并发活动之间进行同步，同时满足时间约束。

- 计算映射：能够控制活动如何映射到底层处理器上，包括将活动固定到核心，或允许活动在处理器内部或不同处理器之间迁移。

- 活动调度控制：为确保并发活动的执行符合可调度性分析所依据的模型，需要能够控制底层平台上的并发调度方式，例如采用抢占式或非抢占式调度，或固定优先级、最早截止时间优先或轮转调度等方法。

为满足这些要求，基本上有两种不同的方法：

- 使用顺序编程语言，由库和操作系统调用满足并发与实时需求：这是最常见的方法之一，采用的是

C [25] 语言和实时操作系统，最终还可采用 POSIX 实时支持[26]。

- 使用支持并发和实时作为语言一等实体的编程语言，例如 Ada [27] 或实时 Java [28]。

本节余下部分将结合 C/POSIX 和 Ada 的示例，简要介绍这两种方法。目标是概述这些方法，而非全面介绍其支持情况。有关详细信息，请参阅 [24] 和相关规范 [26][27] 。

3.1 POSIX 规范中的实时支持

POSIX 规范提供了大量支持实时系统编程的机制。并发活动的规范可以使用 POSIX 线程来编写，如清单 1 所示；这里的示例用于规范一个周期性活动。

```
void *活动          _in_线程     _code(void *arg) {
    long period   _us *ps =  (长 *) arg;
    struct 时间规格结构 next, now;
    clock   _获取时间(CLOCK   _REALTIME, &next);
    当 (1) {
        // 活动代码

        时间规格结构 _add_us(&next, *ps);
        时钟   _nanosleep(CLOCK   _REALTIME,TIMER    _ABSTIME,&next,NULL);
    }
    return 空指针;
}
```

清单 1。周期性活动编程示例。

还请注意，程序使用了时钟_nanosleep函数，并配合实时时钟（CLOCK_REALTIME）以及绝对时间标志（TIMER_ABSTIME）的指定。对于实时系统而言，支持时间概念无疑至关重要。实时系统需要的并非日历时钟，而是精确且以恒定速率推进、不会因插入额外的时钟滴答来反映闰秒（如日历时钟那样）而受到影响的时间基准。控制算法需要定期执行，因此必须保持恒定速率。奇怪的是，CLOCK_REALTIME时间基准并非如此，因为它可能随操作系统墙上时间的变化而改变；因此，POSIX 还提供了CLOCK_MONOTONIC时间基准，旨在满足这一用途[5] [26] （第 B.2.8 节）。

为了实现周期性活动，休眠操作应使用绝对时间，而非相对时间。原因在于，线程可能恰好在调用休眠函数之前被中断。如果休眠时间是相对的，例如休眠 5 秒，那么周期性活动的下一次释放时间就会变得不固定，具体取决于中断发生的时机。

[5] 请注意，Linux 对 CLOCK_MONOTONIC 的实现并未完全遵循 POSIX，因此在 Linux 中应改用 CLOCK BOOTTIME。_

这种时间处理方式还可通过以下可选功能加以补充：使用 CPU 执行时间时钟，用于测量特定线程消耗的 CPU 时间；以及设置执行时间定时器，用于测量线程是否超出其“最坏情况执行时间”并发出通知 [6]。这基于每个进程和每个线程的 CPU 时间时钟（CLOCK_THREAD_CPUTIME_ID），以及创建定时器的可能性：当时间到期时，定时器会调用处理函数（使用信号）（列表 2）。线程最终还可以通过定时器获取时间轮询执行时间，作为检测超时的另一种方法。_

```
void 信号 _处理程序(){
    // 处理执行时间超时
}
//...

结构 sigevent 事件;
计时器_t 计时器 _id;
struct itimerspec 中央处理器 _使用率;

// 处理函数
SIGEV _信号 _初始化(&event, SIGALRM);
信号(SIGALRM, sig _处理程序);

// 创建一个线程计时器
计时器_create(CLOCK _线程 _中央处理器时间 _标识符, &事件, &计时器 _id);

// 设置执行时间
cpu_使用率.it _值.tv _秒 = 0;
cpu_使用率.it _值.tv _纳秒 = exec _时间;
cpu_使用率.it _间隔.tv _秒 = 0;
cpu_使用率.it _间隔.tv _纳秒 = 周期;

// 设置计时器
计时器_设置时间(计时器 _ID, 0, &中央处理器 _使用率，空指针）；
```

列表 2。设置执行时间计时器。

POSIX 规范还提供了一种有限的方法来指定实时属性和调度，基本支持固定优先级和轮转调度，并定义线程优先级（列表 3）。其中还包含一项可选策略，用于实现偶发服务器的概念 [29], ——这种方法会提供预分配的周期性服务器预算，以处理非周期性活动。不过，绝大多数系统并未实现这一策略。

```
POSIX 线程_属性_t custom_属性 _先进先出；int
先进先出_最大值 _prio；struct 调度_param 先进
先出_param;

POSIX 线程_attr_初始化(&custom _attr_先进先出);
POSIX 线程_attr_设置调度策略(&custom _attr_先进先出，调度 _先进先出）；
fifo_param.sched _优先级 = sched _get_优先级 _最大值(调度 _先进先出);
POSIX 线程_attr_设置调度参数(&custom _attr_fifo, &fifo _param);
POSIX 线程_create(&thread _id, &custom _attr_fifo, thread _func,NULL);
```

清单 3。使用 POSIX 管理调度属性的示例。

请注意，系统并不局限于只实现 POSIX 的机制。例如，Linux 内核还实现了 调度_DEADLINE 策略（清单 4），该策略支持一种更动态的实时调度算法——最早截止时间优先，也支持预算（使用恒定带宽服务器 [30] 的一种变体）。此外，规范还定义了 调度_OTHER 类的可能性，在这种情况下，操作系统

[6] 超出最坏执行时间的情况，既可能由错误情形（例如活锁）导致，也可能是由于代码执行时间计算（或估算）不正确所致。

可以实现自己的调度方式，但代价是代码无法在不同操作系统之间保持可移植性。

```
结构体 sched_attr 属性; 属性.sched_policy = SCHED_DEADLINE; 属性.sched_
runtime = 10 * 1000000; // 10 毫秒 属性.sched_period = 属性.sched_deadline = 100
* 1000000; // 100 毫秒 sched setattr(0, &属性, flags);
```

清单 4。在 Linux 中为当前 POSIX 线程指定带预算的截止时间。

POSIX 还支持一些机制（清单 5），用于解决使用互斥锁同步线程时可能出现的优先级反转问题。 [31] 当高优先级线程等待一个由低优先级线程持有的互斥锁，而低优先级线程又因中等优先级的线程而无法执行时，就会发生优先级反转。解决这一问题的常见方法依赖于优先级继承（低优先级线程继承被阻塞的高优先级线程的优先级）或优先级上限（低优先级线程以任何可能请求该互斥锁的线程中的最高优先级执行）。尽管 POSIX 规定了优先级继承协议和优先级上限协议，但大多数实现只支持前者。

```
POSIX 线程_互斥锁属性 _t 属性;
POSIX 线程_互斥量_t m;
POSIX 线程_互斥锁属性 _setprotocol(&attr, PTHREAD _PRIO_INHERIT);
POSIX 线程_互斥量_初始化 (&m, &attr);
```

清单 5。指定具有优先级继承的互斥锁。

关于将计算映射到底层处理器的能力，尽管这并非 POSIX 规范的一部分，许多 pthread 库实现都提供了机制，可通过设置亲和性来指定线程可以执行的核心（CPU）集合。清单 6 给出了一个示例：执行这段代码的线程只能在处理器的核心 0 和核心 1 上执行。

```
中央处理器_设置_t cpuset; POSIX 线程_t thread; //... 线程 =
POSIX 线程_self(); CPU_ZERO(&cpuset); for (int i = 0; i < 2; i++)
CPU_SET(i, &cpuset); pthread setaffinity np(thread, sizeof(cpuset), &
cpuset);
```

清单 6。Linux 中不可移植的亲和性设置。

将顺序语言与库和实时支持结合使用会带来若干问题[24]。首先，编译器（甚至操作系统）不了解不同线程之间的并发，因此大多数并发错误都难以检测，尤其是在编译阶段。此外，由于应用程序的并发特性以及可能发生的通信/同步无法直接体现，代码的可读性会受到严重影响。另外，正如前文所述，即使已有规范，程序在不同操作系统之间也往往无法移植。

*3.2 Ada*语言中的实时支持

另一种方法是使用一种尽可能将并发和实时概念作为一等语言实体支持的语言。Ada语言就是其中一个例子 [27]。Ada标准规定了一组核心机制，其中第9章涉及并发；此外还规定了一组可选提供的附录。具体而言，附录 D 规定了一组支持实时编程的机制。

Ada语言通过一种名为任务的语言类型，为并发活动提供语言级支持。任务对象是逻辑上自主的并发活动，具有独立的状态。只要保证并发属性，语言便不强制规定运行时如何将任务映射到底层线程[32]（第9章，第11段）：

“并发任务执行可以在多计算机系统、多处理器上实现，也可以通过在单个物理处理器上交错执行来实现。另一方面，只要某种实现能够确定：由不同物理处理器并行执行给定任务的部分执行过程可以达到所要求的语义效果，它就可以选择采用这种方式。”

该语言提供了用于指定周期性和非周期/偶发任务的模型，不过仍需用户编程（列表7给出了一个周期性任务的示例）。在此示例中，也可以看到实时基准时间的用法。该语言在规范中同时提供了日历墙钟时间基准和单调时钟（详见专门介绍实时机制的附录 D）。通过延迟至这一语言关键字支持绝对时间。

```
任务体 Cyclic 为 Next_Period : Ada.Real_Time.Time := 首次_释放; Period : Ada.Real_
Time.Time_Span := Ada.Real_Time.Milliseconds(50); -- 声明开始 -- 初始化循环 -- 任务代码
Next_Period := Next_Period + Period; 延迟 直到 Next_Period; 结束 循环; 结束 Cyclic;
```

列表7。Ada 周期性任务。

```
受保护类型 有界_缓冲区 是
    入口 获取 (X : 输出 项目);
    入口 放入 (X : 输入 项目);
私有
    Get_索引: 缓冲区_索引 := 1;
    Put_索引: 缓冲区_索引 := 1;
    计数: 缓冲区_计数 := 0;
    数据: 缓冲区_数组;
结束 有界_缓冲区;

受保护主体 有界_缓冲区 是
    条目 获取 (X : 输出 项目) 当 计数 > 0 是
    开始
        X := 数据(获取_索引);
        Get_索引 := (获取_索引 模 最大值_缓冲区_大小) + 1;
        计数 := 计数 - 1;
    结束 获取;
    条目 放入 (X : 进入 项目) 当 计数 < 最大值_缓冲区_大小为
    开始
```

```
            数据（放入_索引）：= X;
            Put_索引: = （放入 _索引 取模 最大值 _缓冲区 _大小）+ 1;
            计数 := 计数 + 1;
        结束 放入;
结束 有界 _缓冲区;

My_缓冲区: 有界 _缓冲区;
```

清单 8。使用 Ada 保护对象实现的有界缓冲区。

受保护对象支持数据共享和同步。受保护对象是一种实现监视器概念的语言类型 [33]。清单 8 展示了如何使用受保护对象定义一种支持有界缓冲区（可安全并发使用）的类型；清单 9 则展示了如何使用受保护对象实现偶发任务的释放机制。

```
受保护的事件是
   入口等待;
   过程信号;
私有
   已发生: 布尔型: = 假;
结束 事件;

受保护 体 事件 是
    入口 等待 当 已发生 是
    开始
        已发生 := 假;
    结束 等待;
    过程 信号 是
    开始
        已发生 := 真;
    结束 信号;
结束 事件;

任务 主体 释放 是
开始
    -- ...
    如果 某个 _条件 那么
        事件信号;
    结束 如果;
    -- ...
结束 释放;

任务 主体 偶发 是
开始
    -- ...
    Event 等待;
    -- 代码
结束 偶发;
```

清单 9. 使用 Ada 保护对象进行任务同步。

在实时附录（附录 D）中，Ada 语言规范提供了非常完整的语言特性和库，旨在支持实时编程 [32]。该附录定义的调度模型允许在同一个应用程序中使用不同的调度策略：

- 先进先出_优先级内_优先级 – 在每个优先级内，任务按照先进先出的顺序处理。任务可以抢占优先级较低的任务。
- 非_抢占式_先进先出_优先级内_优先级 – 在适用的每个优先级内，任务会一直运行至完成，除非被阻塞或执行延迟语句。任务不会被优先级更高的任务抢占。
- 轮询_调度_范围内_优先级 – 在每个优先级级别内，任务按可指定的时间间隔进行分时片处理。

- 最早截止时间优先_跨越_优先级 – 这提供了最早截止时间优先调度。其基本思路是，在一系列优先级级别中，每个任务都有一个截止时间，并优先处理截止时间最早的任务。

由于这些策略应用于优先级（或优先级范围），因此可以定义分层调度（第一层基于优先级）[34]。例如，清单 10 定义了较高优先级（11 到 24）采用固定优先级调度，优先级范围为 2 到 10 的任务采用动态调度，而后台任务则全部置于最低优先级，并采用时间片轮转。

```
pragma 优先级 _特定 _调度(FIFO _优先级内_优先级,11,24);
pragma 优先级 _特定 _调度（最早截止时间优先_跨 _优先级，2，10）；
pragma 优先级 _特定 _调度（轮 _询 _优先级内_，1,1）；
```

列表 10。分层调度。

由于支持固定优先级和动态优先级，该语言还允许指定任务的属性，例如优先级和截止时间（列表 11）。

```
任务 固定优先级调度
   task
   with 优先级 => 20;

任务 最早截止时间优先
   task
   with 优先级 => 10,
        相对 _截止时间 => Ada.Real _Time.Milliseconds(50);

受保护的 事件
   带有优先级 => 20
```

列表 11. 优先级和截止时间属性的规范。

如清单 11 中所述，该语言还支持为受保护对象分配优先级，以便通过即时天花板协议，结合栈资源策略 [35], 或多处理器资源共享协议（MrsP）[36]，实现任务间无优先级反转的数据通信与同步。所有任务都会被分配优先级（对于动态优先级区间中的任务，优先级定义了抢占级别），而受保护对象的优先级可以设置为优先级上限（所有使用该对象的任务中的最高优先级）。任务一旦进入受保护对象，其活动优先级便会提升至该优先级上限。在此示例中，受保护对象 `Event` 将用于同步 `FPS_`任务 和 `EDF_`任务，因此其优先级至少应达到这两个任务中较高的优先级。

```
受保护的 最坏情况执行时间_超时 是
    入口 等待;
    过程 触发;
私有
    已发生: 布尔型: = 假;
结束 最坏情况执行时间_超时;
受保护 主体 最坏情况执行时间_超时 是
    入口 等待 已发生时 是
    开始
        已发生: = False;
    结束 等待;
    过程 触发 是
    开始
        已发生: = True;
    结束 信号;
结束 最坏情况执行时间_超时;
任务 主体 周期性 是
    Ex_计时器: Ada.Execution_Time.Timers.Timer(Current_任务);
开始
    loop
        Set_处理程序(Ex_计时器,任务_最坏情况执行时间,
                     WCET_超时.触发'Access);
        选择
            WCET_超时.等待;
            -- 错误 recovery
        然后中止
            -- 任务代码
        结束 选择;
        取消_处理程序(Ex_计时器, Cancelled);
        下一项 := 下一个 + 周期;
        延迟至 Next;
    结束 循环;
结束 周期性;
```

清单 12。使用执行时间定时器检测最坏情况执行时间超限。

该附录还提供了对任务执行时间进行控制的支持（清单 12），包括为任务和中断指定执行时间定时器，以及设置任务组预算。不过，除面向研究的 Marte OS 操作系统 [37]所提供的实现外，这些机制很少有完整实现可供免费使用。清单 12 中的方法会在计时器到期时异步调用 最坏情况执行时间_超时.触发；另一种方法是使用 时间_剩余函数检查是否超时。

该附录提供了将多处理器实现划分为互不相交的分派域的能力，使每个域都拥有独立的调度与分派队列。这样一来，便可以将任务固定到核心，或通过将任务集群划分到各个域中来设置有限的迁移方法 [38]（清单 13）。

也可以在执行期间动态地将任务分配给 CPU，例如根据调度条件，在经过一定执行时间后迁移任务 [39]（列表 14）。

```
-- CPU 从 1 开始
-- 此代码假定多处理器中有 8 个 CPU
集群_1: 调度_分派域 := Create(1, 4);
集群_2: 调度_分派域 := Create(5, 8);

任务 T1 with 调度_分派域 => 集群_1;

-- 其他任务
```

清单 13。基于集群的调度。

```
如果 释放 _时间 >= 时隙 _开始 和 释放 _时间 < 结束 _of_阶段 _1 然后
    Set_中央处理器（客户端 _阶段 _1_中央处理器，当前 _任务）;
    切换 _计时器。设置 _处理程序(结束 _of_阶段 _1, 处理程序'Access);
    客户端 _当前 _阶段 := 阶段 _1;
    已释放 := True;
elsif 释放 _时间 >= _开始 _of_阶段 _2
                    and 释放 _时间 < 结束 _of_时隙 然后
    Set_中央处理器（客户端 _阶段 _2_中央处理器，，当前 _任务）;
    切换 _计时器.设置 _处理程序(结束 _of_时隙, 处理程序'访问);
    客户端 _当前 _阶段 := 阶段 _2;
    已释放 := 是;
else
    客户端 _当前 _阶段 := 未 _已释放;
    切换 _计时器.设置 _处理程序(开始 _of_阶段 _2, 处理程序'访问);
结束 if;
```

列表 14。任务拆分调度算法的实现。

Ada语言规范还包含许多其他用于并发和实时处理的机制。尽管如此，在20世纪90年代末，人们普遍认为，Ada的并发机制虽全面但较为复杂，可能会妨碍在关键系统中使用该语言模型。人们认识到，这些系统也需要简单且可认证的并发机制。具体来说，以下几点至关重要：

- 移除开销较高的机制，以提高实时内核的效率。
- 简化实时内核并移除无法认证的机制，以便验证高完整性系统。
- 通过移除非确定性机制，降低应用程序的非确定性。

为了解决这一问题，Ada 实时工作组提议定义一个受限的并发机制配置，由此形成了 Ravenscar [7] 配置 [40], ，并在该语言的 2005年修订版中实现了标准化。此配置定义了一种基于固定优先级调度的模型，该模型在高完整性/安全关键行业中广受认可，并从语言中移除了所有动态并发机制。最近又定义了一种限制稍宽松的配置，即 Jorvik 配置 [41], ，但它仍然保持了相同的并发模型。

使用支持并发和实时的语言，能显著提高应用程序的可读性和正确性；这对于信息物理系统至关重要，对于关键应用程序更是如此。不过，这种做法也存在一些缺点 [24]。其一，应用程序的不同组件往往使用不同的语言；如果采用操作系统规范，这种情况会更容易处理。例如，如果 Ada 运行时基于 POSIX 接口实现，而 C 代码也使用同一接口处理并发，那么链接并发 Ada 代码和 C 代码就容易得多。其二，需要在不同的平台和操作系统上实现语言运行时。这意味着硬件或软件平台问世后，所需的语言支持可能要过一段时间才能提供 [8]。

[7] 尽管 Ravenscar 起初也被当作一个首字母缩略词，但这个名称源自英格兰北部的一座小城，定义该配置初始版本的国际实时 Ada 研讨会曾在那里举行。[8] 这也是 Ada 语言难以在关键系统领域之外得到应用的原因之一：将语言运行时移植到种类繁多的现有处理器和操作系统上，成本很高。

4. 并行实时系统编程

从并发模型迈向兼顾并发与并行性的模型时，实时系统编程不仅需要考虑上述问题，还要考虑并发活动中的顺序代码可能会并行执行这一事实。因此，编程模型必须兼具两种能力：既能定义应用程序固有的并发活动，也能利用这些活动的算法所蕴含的潜在并行性。

此外，正如第2.2小节所述，线程调度还需要结合并行计算到调度实体的映射。虽然这看起来可能是运行时和操作系统的问题，但当并行实体和并发实体在编程模型中得到指定时，也需要相应的机制来指定底层运行时和操作系统应如何管理映射与调度（如图4所示的集成垂直栈）。与前面几个小节的讨论类似，这可以通过库和对底层运行时的调用来实现，也可以直接在编程模型层面提供支持。

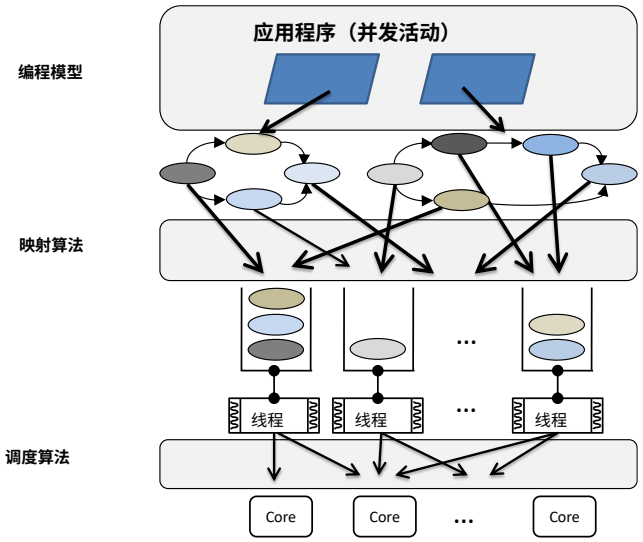


图4：并行计算映射与调度的垂直技术栈（改编自[14]）。

并行实时系统的可编程性仍是一个研究与开发课题。尽管已有少数几种方法，但尚无完整模型（更不用说稳定的产品了）。本文接下来的部分将概述目前正在研究和开发的两种现有方法：

- 第 5 节介绍了基于 OpenMP [42], 任务模型的一个子集的使用。
- 第 6 节介绍了与 Ada 语言实时特性相结合的 Ada 2022 [43] 并行编程模型。

尽管这两种方法都基于面向实时系统的类似细粒度并行模型，但它们来自两个不同的社区，发展方向也截然相反，这反映在主要的设计差异以及整合并行性与实时

-时间。OpenMP 源自高性能计算社区，支持一套非常完善且复杂的并行编程模型。另一方面，Ada 是一项主要由关键系统与实时系统社区开发的技术，

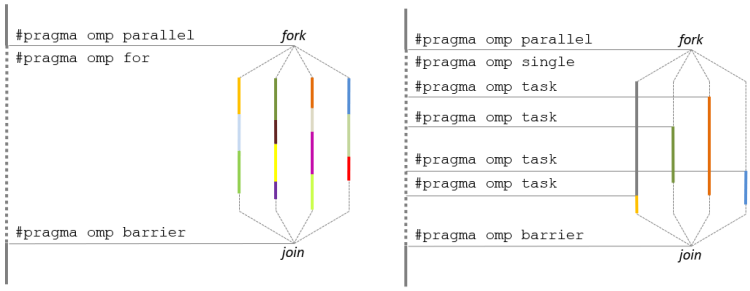
为实时系统编程提供全面而复杂的支持。将实时性与并行性结合起来意味着：(i) 在 OpenMP 中为并行模型加入实时性；(ii) 在 Ada 中为并发实时语言加入并行编程模型。

除这两种方法外，其他技术中将实时性与并行性结合起来的研究寥寥无几。[44] 提出了一个初步方案，将 Java fork//join 模型 [45] 与 Java 实时规范 [28]，相结合；而 [46] 则对嵌入式多核构件 [47]，、英特尔线程构建模块 [48] 和高性能 ParalleX [49] 框架中的实时执行进行了初步分析。然而，据作者所知，这些研究并未继续推进。

5. OpenMP 任务模型

用于展示其在实时应用中用途的框架之一，是 OpenMP 任务模型 [42] （第 12 章），其中通过数据依赖定义优先约束。OpenMP 是高性能计算领域最常见的并行编程模型之一，也日益应用于嵌入式计算系统，以及关键计算系统中的高性能应用程序（例如自动驾驶中的视频处理）。

OpenMP 最初采用线程中心模型，以广泛利用数据并行和循环密集型应用程序。但自 OpenMP 3.0 起，该规范逐渐演变为任务中心模型，以支持非常复杂的非规则细粒度并行性。OpenMP 4.0 引入了数据依赖，旨在通过任务图实现数据流计算模型。该模型使程序员能够定义隐式任务和显式任务，以及它们之间的数据依赖。任务在运行时由线程团队执行，并由底层运行时管理，从而将复杂性对程序员隐藏起来。



(a) (b)

图5。具有 (a) 结构化并行性和 (b) 非结构化并行性的分叉/汇合模型 [14]。

5.1 对实时并行模型的支持

OpenMP 程序以单个执行线程启动，称为初始线程。并行性通过并行构造实现。遇到此类构造时，会生成一个线程团队。遇到屏障时，这些线程会汇合。因此，OpenMP 任务模型最简单的形式

完全支持分叉/汇合计算模型（图5）。在并行区域内，可通过不同的构造实现并行性：`for`、`section` 或 `task`。`for` 和 `section` 提供结构化并行，而任务模型则通过非结构化并行实现充分的灵活性。

OpenMP 任务模型的全部灵活性通过嵌套任务构造以及任务间的数据依赖来实现。在这种情况下，可以使用 OpenMP 任务模型对第2小节中介绍并在清单15和图6中展示的并行有向无环图 (DAG) [9] 进行编程。

为了理解 OpenMP 任务编程模型与 DAG实时模型之间的映射，一个重要概念是对 OpenMP任务调度点 (TSP) [42] （第12.9节）的定义。TSP 是程序中的特定情形，在这些情形下，任务可以暂停，而执行该任务的线程可以被分配去执行另一个任务。任务创建和完成时，以及在 `taskwait` 指令和显式、隐式屏障等同步点处，都会出现 TSP。TSP 将任务区域划分为若干任务部分，这些部分（从 OpenMP 的角度来看）会不间断地执行。同一任务区域的不同部分按照遇到它们的顺序执行，因为它们是顺序执行的。

清单15/图6所示的示例标明了每个任务区域划分成的各个部分：`T0`由`tp00`至`tp04`组成；`T1`由`tp10`和`tp11`组成；`T2`、`T3`和`T4`各自只由一个部分组成。当任务的执行到达 TSP 时，OpenMP 运行时可以继续执行当前任务，也可以恢复先前挂起的任务。运行时如何作出这些决定由实现决定，但必须满足任务调度约束（TSC）[42] （第12.9节）：

1. 新绑定任务的调度受到当前绑定到该线程且未暂停在屏障区域中的任务区域集合的限制。如果该集合为空，则可以调度任何新的绑定任务。否则，只有当新绑定任务是集合中每个任务的后代任务时，才可以调度它。这将在第 5.3 小节中进一步讨论。[10]。2. 依赖任务必须等到其任务依赖得到满足后才能开始执行。3. 如果与某个任务互斥的任务已被调度但尚未完成，则不得调度该任务。4. 如果通过包含 if 子句的构造生成显式任务，且该子句中的表达式计算结果为 false，并且已满足前述约束，则该任务会在生成后立即执行。

[9] 最新的 OpenMP 规范还提供了 Variant子句，使其能够映射到实时系统中较新的有向无环图模型，这些模型面向异构平台。[10]在较早版本的 OpenMP规范中，这一条款称为 TSC2，因此许多文献都使用这个缩写来指代它。

```
#pragma omp parallel {
#pragma omp single                                                    // T0
{
        ...                             // tp00
        #pragma omp task depend (out: x)                              // T1
        {
                ...                     // tp10
                #pragma omp 任务 if(false)                            // T4
                { ... }                 // tp4
                ...                     // tp11
        }
        ...                             // tp01
        #pragma omp task depend (in: x)                               // T2
        { ... }                         // tp2
        ...                             // tp02
        #pragma omp task等待
        ...                             // tp03
        #pragma omp 任务                                              // T3
        { ... }                         // tp3
        ...                             // tp04
}}
```

清单15。OpenMP 结构示例（改编自 [14]）。

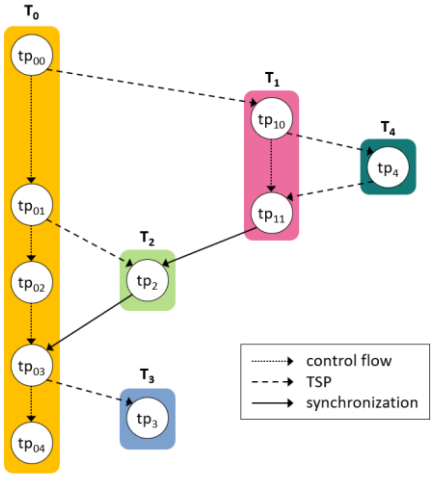


图6。有向无环图示例（改编自 [14]）。图6中的有向无环图由以下方式创建：

- 同一任务各部分之间的控制流依赖（例如，只有在tp00执行后，才能执行tp01）。

- 任务创建：由于T1、T2和T3由T0创建，因此tp10只能在tp00之后执行，tp2只能在tp01之后执行，而tp3则在tp03之后执行；T4由T1创建，因此tp4在tp10之后执行。

- 任务完成：由于 if 子句中的 TSC 会强制T4在创建后立即执行，这意味着tp11会等待tp4。

- 由于数据依赖而产生的 TSC 同步：T2依赖变量x（in），而T1会写入该变量（out），因此tp2依赖tp11。

- 由于屏障而产生的同步中，taskwait 指令意味着 tp03 需要等待 T2 (tp2)、T1 (tp11) 和 T4 (tp4) 完成。由于 tp2 依赖于 tp11，而 tp11 依赖于 tp4，因此只需要保留 tp2 与 tp03 之间的箭头。

可以采用一种简化方式，使用受限模型，仅通过数据依赖来指定有向无环图的边。在这种情况下，程序将所有任务置于同一层级（单个指令的初始任务除外），并且任务之间仅通过任务的 `depend` 子句进行同步（列表 16 和图 7）。

```
void Cholesky分解(float *M, int ntiles) {
    for (k = 0; k < ntiles; k++) {
        #pragma omp 任务 depend (out:M[k][k])
        Cholesky 分解 (M[k][k]);
        for (i = k + 1; i < ntiles; i++)
            #pragma omp 任务 depend (in:M[k][k]) depend (out:M[k][i])
            trsm（M[k][k]，M[k][i]);
        for (i = k + 1; i < ntiles; i++) {
            for (j = k + 1; j < i; j++)
                #pragma omp task depend (in:M[k][i],M[k][j])
                                    depend（输出:M[j][i]）
                通用矩阵乘法 (M[k][i]， M[k][j]， M[j][i]);
            #pragma omp 任务 depend (in:M[k][i])
                                    依赖 (out:M[i][i])
            对称矩阵秩-k 更新 (M[k][i]， M[i][i]);
        }
    }
}
```

列表 16。仅包含数据依赖的结构

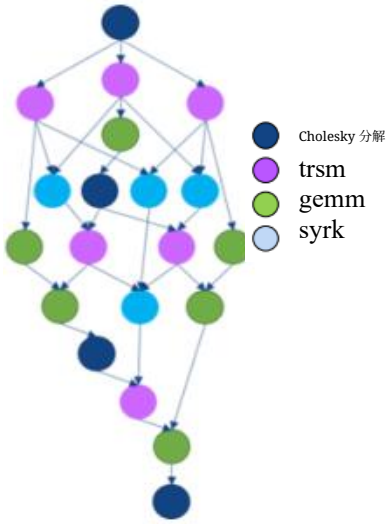


图 7。具有数据依赖的 OpenMP 有向无环图

即使进行了这种简化，OpenMP 有向无环图模型在开发并行实时模型方面仍提供了显著的灵活性。然而，这种灵活性也给程序员带来了额外负担：需要在任务之间构建安全的数据共享与同步机制，而这本来就是在 OpenMP [50] 中需要考虑的难题。

请注意，为了支持离线实时分析，必须在离线时已知完整的有向无环图。因此，需要提取所有会影响并行执行的信息，即实例化依赖于循环迭代和/或 if 条件的任务和数据依赖关系。如果无法离线完成这一点，则需要考虑悲观行为假设，从而进行悲观的可调度性分析 [50][51]。

尽管支持使用实时并行模型，当前 OpenMP 规范并不支持实时调度属性，例如截止期限、时间基准或最坏执行时间（WCET）。此外，尽管支持任务优先级这一概念，其语义却非常宽松(该子句规定的是一种“提示” [42] (第 12.4 节)，因此无法保证实时行为。

有人提议 [52] 为关键实时系统定义一种 OpenMP 配置，其中包括：

- 收紧任务优先级子句的语义，从而支持优先级驱动的任务调度算法。
- 为任务定义截止时间属性，从而支持动态调度算法。
- 规定任务事件子句，使任务能够根据时间基准或偶发事件周期性释放。

该提议未获考虑，决定留待日后处理；目前，实时行为仍属于一种实现方式。[52], 对此作了简要介绍，并讨论了 OpenMP 为不同并行区域指定独立线程团队的各种方案；我们将在第 5.3 小节讨论这一问题。

5.2 通信与同步

OpenMP 提供了一小组机制，以便在并行任务之间安全地共享数据。`depend` 子句就是其中一种机制，因为它允许实现保证任务之间并行读写的顺序，并确保互斥。不过，如果目标只是在访问共享数据时保证互斥，OpenMP 还提供了诸如 `critical` 构造 [42]（第 15.2 节）或 `locks` [42] （第 18.9 节）这样的机制。从实时角度来看，由于 OpenMP 不支持指定实时属性，这些机制不会实现任何形式的优先级继承或优先级上限 [11]。

OpenMP 运行时的实现可以将这些机制映射到底层的 POSIX 互斥锁，并使用实时扩展，如第 3.1 小节所述。然而，在编程模型层面，没有任何机制允许程序员控制这种行为。若要正确使用这些机制，就需要在 OpenMP 运行时内部进行编程。

5.3 映射与调度控制

OpenMP 任务模型采用两级映射与调度方法，类似于此前图 4 中所示的方法。程序中指定的 OpenMP 任务会映射到并行区域的线程团队，再由操作系统执行。

[11] 虽然这与实时无关，但在 OpenMP 任务中使用 critical 构造和锁时需要精心设计，因为这些机制作用于线程级别。对于可以在线程之间迁移的非绑定任务来说，这一点尤其重要。

至于任务到线程的映射，规范也将其留给实现决定。典型的实现采用广度优先（广度优先搜索）和工作优先（工作优先调度）方法：广度优先会先创建所有子任务再执行，而工作优先会在创建新任务后立即执行这些任务（理论上更优，因为它能改善缓存局部性）。有些实现对所有任务（位于同一线程团队中）使用单个队列，但较新的方法（例如 LLVM 实现）采用工作窃取策略 [19]:

- 每个线程都有自己的队列，创建的任务会插入其线程队列的队首。
- 线程首先尝试执行自己队列中的任务（优先从队首取出，以提高缓存局部性）。
- 如果某个线程的队列为空，它就会从另一个线程的队列中移除（窃取）任务，但这次从队尾取出，以减少争用和对缓存造成影响的可能性（取出的是较旧的“冷”任务）。

OpenMP 区分绑定 [12]（默认）和非绑定 [13]任务。绑定任务必须由同一个线程执行，而非绑定任务则可以在任务同步点从一个线程迁移到另一个线程。非绑定任务提供了一种工作保持策略，线程可以暂停当前任务的执行，转而执行另一个任务。根据任务类型，如果有可用线程，暂停的任务可以在同一线程或不同线程上恢复执行。这保证只要有工作要做，就不会有线程处于空闲状态。这是实时系统调度中的一个重要概念，因为它可以进行更简单、保守性更低的分析 [53]。此外，针对工作保持策略，也存在实时友好的工作窃取实现 [18]。

相反，由于第 5.1 小节提出的第一项任务调度约束，使用绑定任务可能导致非工作保持型任务调度。该约束源于 OpenMP 以线程为中心的规范与以任务为中心的规范之间的权衡。由于许多 OpenMP 机制都在线程级别工作（ critical 构造、线程私有变量），允许任务在线程之间迁移可能会引发并发错误，因此默认情况下不允许这样做。此外，创建绑定任务时，只能将其映射到空闲线程，或映射到所有任务都是该新任务祖先的线程。尽管已有针对这种方法的分析（例如 [54]），但其结果偏保守，在某些情况下，响应时间甚至超过图的总执行时间（因此比顺序执行更差）。请注意，使用绑定任务并采用工作优先调度（WFS）映射方法会导致顺序执行。

OpenMP 也没有规定团队线程如何进行调度 [14]。这使得在编程层面无法提供任何实时调度控制。此外，该规范将两个并行区域中的线程团队视为完全独立，彼此互不知晓。尽管许多实时研究都考虑单个有向无环图的可调度性分析，但应用程序可能需要在多个并发活动中实现并行性。

[12] 绑定任务的所有任务部分都必须由同一个线程执行。 [13] 非绑定任务的各个任务部分可以由不同线程执行。 [14] 大多数实现会将 OpenMP 线程与操作系统线程一对一映射，例如 POSIX 线程。

这一点在 [52], 中有所提及，其中提出了使用嵌套任务或嵌套并行区域的替代方案，但这些方案依赖于能够指定实时优先级或截止时间，从而让 OpenMP 运行时全面负责任务和线程的调度（清单 17）。

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task 优先级 (p1) // T1：OpenMP–有向无环图1
    { RT 任务 1 ( ) }
    #pragma omp task 优先级 (p2) // T2：OpenMP–有向无环图2
    { RT 任务 2 ( ) }
    // ...
    #pragma omp task priority (pn) // Tn：OpenMP–有向无环图n
    { 实时任务 n ( ) }
}
```

清单 17。实时 OpenMP 提案 [52]。

另一种可行的方法是利用这样一个事实：OpenMP 的实现会将线程的调度委托给底层操作系统。这样，实时行为便可由底层实时操作系统提供。在这种情况下，实时并发活动可以映射到不同的实时线程（例如，使用 POSIX 线程），这些线程具有不同的优先级，且每个线程执行一个不同的有向无环图（清单 18）。这种方法还有一个额外优势：只要所有组件使用相同的底层线程模型，就可以使用不同语言开发应用程序。

```
void *T1(void *arg) {
    // T1：OpenMP–有向无环图1
    #pragma omp parallel {
    #pragma omp 单线程 {
        // ...
    } }
    return 空指针;
}

无类型 *T2(无类型 *arg) {
    // T2 : OpenMP–有向无环图2
    #pragma omp parallel {
    #pragma omp 单线程 {
        // ...
    } }
    return 空指针;
}

// ...

void 主函数(){
//...

POSIX 线程_attr_初始化(&custom _attr_fifo1);
POSIX 线程_attr_设置调度策略(&custom _attr_fifo1, 调度 _先进先出);
fifo_param1.sched _优先级 = p1;
POSIX 线程_attr_设置调度参数(&custom _attr_fifo1, &fifo _param1);
POSIX 线程_create(&thread _id1, &custom _attr_先进先出1, T1, 空指针);

POSIX 线程_attr_初始化(&custom _attr_先进先出2);
POSIX 线程_attr_设置调度策略(&custom _attr_fifo2, 调度 _先进先出）;
fifo_param1.sched _优先级 = p2;
POSIX 线程_attr_设置调度参数(&custom _attr_fifo2, &fifo _param2);
POSIX 线程_create(&thread _id1, &custom _attr_fifo2, T2, 空指针);
// ...
```

清单 18。混合使用 POSIX 线程与 OpenMP。

不过，要使这种方法奏效，底层的 OpenMP 运行时实现必须将创建并行区域的线程的优先级传递给所创建线程团队中的其他线程，

并行区域。UpScale 框架的实现也采用了类似的方法 [55]„ 它实现了一个轻量级的 OpenMP 运行时 [56] ，将线程优先级传递给团队线程以及底层操作系统（此处为 Erika OS [57]）。该实现还将任务调度点的信息传递给操作系统。这使得操作系统能够采用有限抢占式调度方法 [58] ，从而减少开销并提高缓存局部性 [14]。

此外，还需要能够控制线程与底层处理器之间的亲和性。OpenMP 包含 `proc_bind` 指令，允许程序员指定将线程分配到核心的策略。借助所提供的机制，可以设定线程团队中的所有线程绑定到同一个核心、绑定到连续的核心，或分散到不同分区（集群）的核心上。因此，可以定义全局调度或分区调度，但无法采用集群调度或任务分割等更灵活的方法 [38][39]。同样，这些功能需要由特定实现提供。

在实时系统中使用 OpenMP 任务模型仍处于研究阶段。[52], 中已有部分功能的有限实现，但仍需开展更多工作，尤其是如何在同一应用程序中集成多个并行实时活动，以及如何控制任务到线程的映射，以便进行更可预测（且不那么悲观）的分析 [17]。

6. Ada 2022

如第 3.2 小节所述，Ada 编程语言为并发和实时系统提供了广泛支持。作为这项支持的补充，该语言即将推出的修订版[43] 还纳入了细粒度并行模型 [59]。

在此模型中，`parallel`关键字用于标示可由多个控制线程并行执行的语言结构 [43] （第 5.1 节），这些结构既包括`block`语句 [43] （第 5.6.1 节）（清单 19），也包括循环 [43] （第 5.5 节）（清单 20 和清单 21）。该规范还提供了对循环分区为多个块以及变量归约的控制（清单 22 ）。

通过改变 Ada 任务的概念 [43] （第 9 节），将并行模型与 Ada 既有的并发模型集成起来，使 Ada 任务在执行`parallel`结构时，能够表示多个可并行运行的逻辑控制线程。因此，可以在任务体中使用并行块和循环（代码清单 23）。

```
过程 遍历（T: 表达式                    _指针）为
开始
   如果 T /= 空指针 且 则 T.all 属于 二元              _运算'Class 则
      并行 执行
         遍历 (T.Left);
      and
         遍历 (T.Right);
      and
         Put_Line("处理能力" &
                  Ada.Tags.Expanded_Name (T'Tag));
      结束 do;
   结束 if;
结束 遍历;
```

清单 19。并行块示例 [60] （第 5.6.1 节）。

```
并行 for 循环 I in Grid'Range(1) 循环
    Grid(I, 1) :=
        (对于 所有 J 在 Grid'Range(2) => Grid(I,J) = True);
结束 循环;
```

清单 20。并行循环示例 [60] (第 5.5 节)。

```
并行 for Element of Board 循环
    元素: = 元素 * 2.0;
结束 循环;
```

列表 21。并行迭代器示例 [60] (第 5.5.2 节)。

```
声明
    子类型 数据块    _数值 为 自然数 范围 1 .. 8;
    部分     _Sum,
    部分     _部分最大值: 数组 (数据块 _块编号) 的 自然数 :=
                 (其他 => 0);
    部分     _最小值 : 数组 (数据块    _块编号) 的 自然数 :=
                 (其他 => 自然数'Last);
开始
    并行（块（在 块                  _编号）
    对于 I 在 Grid'Range(1) 循环
        声明
            True_计数: 常量 自然数: =
                [对于 J 在 Grid'Range(2) =>
                  (如果 Grid (I, J) 则 1 否则 0)]'Reduce("+",0);
        开始
            部分     _Sum (数据块) := @ + True    _计数;
            部分     _最小值（数据块）: = 自然数'Min(@, True   _计数);
            部分     _最大值（数据块）: = 自然数'Max(@, True   _计数);
        end;
    结束 循环;
    Put_Line
      ("总计=" & 部分            _Sum'Reduce("+", 0)'Image & ", 最小值=" &
       部分      _最小值'Reduce(自然数'Min, 自然数'Last)'Image &
       ", 最大值=" & 部分       _Max'Reduce(Natural'Max, 0)'Image);
end;
```

清单 22。使用块和归约的并行循环示例 [60] （第 5.5 节）。

```
函数 Sum(T: 某个          _数组  _T; 首次, 最后: 某个      _索引  _T)
    返回 整数 是
    S : 整数 := 0;
开始
    并行 (Num      _CPU)
    for I in 首次 .. Last when I mod 2 = 1 循环
        原子    _Add (S, I);
    结束 循环;
    return S;
结束 Sum;

任务 body Cyclic is
   Next_Period : Ada.Real      _时间.时间 := 首次          _释放;
   周期 : Ada.Real          _时间.时间  _时长 :=
                Ada.Real  _Time.Milliseconds(500);
   T: 某种  _数组    T;
   S, S_L, S_R: 整数;
开始
   -- 初始化
   loop
      延迟 直到 下一个    _周期;
      -- ...
      并行 执行
         S_L := Sum(T, T'首次, T'首次 + T'Length/2 - 1);
      and
         S_R := Sum(T, T'首个 + T'长度/2, T'最后一个);
      结束 执行;
      S := S _L + S _R;
      -- ...
      Next_周期 := 下一个    _周期 + 周期;
   结束 循环;
结束 Cyclic;
```

代码清单 23。具有细粒度并行执行的任务。

6.1 对实时并行模型的支持

即将推出的 Ada 语言版本提供的并行机制，仅支持分叉/汇合并行模型。每个并行构造都会将计算拆分为一组并行控制线程，这些线程必须终止，该构造才能终止。图 8 展示了代码清单 23 在使用 4 个 CPU 时对此模型的实例化。

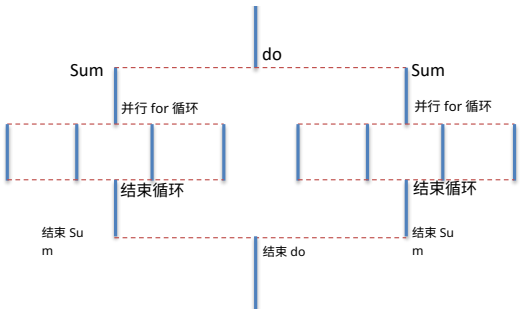


图 8。Ada 并行计算的分叉/汇合模型。

虽然也考虑过一种更通用的模型[59], , 但 Ada 设计背后的基本理念是，让编译器尽可能多地在编译时发现错误，而不是等到这些错误在运行时出现，从而尽可能减轻程序员构建安全并发（及并行）程序的负担。最终决定，为支持并行性而添加到语言中的新能力，也应让编译器能够发现尽可能多的此类问题，同时减少

性能损失。这使得模型被限制为分叉/汇合，同时引入了新的注解，用于指定全局数据的使用方式以及阻塞操作[61]。性能会有所牺牲，但只会发生在非常特殊的情况下：允许并行循环中“未-汇合”的循环迭代（例如波前）可带来更快的处理速度[62]。

Ada 并行模型的开发保留了与 OpenMP规范有限兼容的可能性，即 Ada 模型可以基于 OpenMP 运行时实现 [62][63]。这样一来，便可以使用不同语言（如 C、C++ 和 Ada）编写应用程序，并且都使用相同的 OpenMP 并行支持。它还允许 Ada 程序调用底层运行时，采用非结构化并行，但代价是超出 Ada 的安全范围。

在这种集成方案中，没有并行性的 Ada任务映射到一个操作系统线程，而包含并行构造的每个 Ada任务都与 OpenMP 提供的线程池相关联，以执行细粒度并行（图 9）。此外，该方案假设每个 OpenMP 线程都一对一映射到操作系统线程，这通常确实如此。这简化了分析和实现，也有助于在操作系统内保持一致的调度。

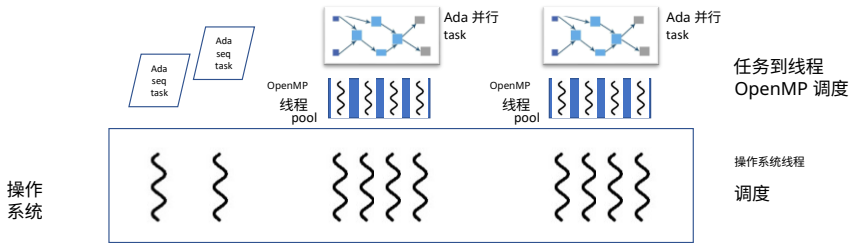


图 9。Ada 到 OpenMP 的映射。

由于 OpenMP 中多个并行团队彼此独立且缺乏语义定义（第 5.3 小节），这种方法受到限制，因此仍是一个有待研究的领域。另一种方法采用模板化执行，强制所有 Ada 任务都访问唯一的 OpenMP 线程团队 [64], ，但这需要显式修改程序，并且可能遭遇优先级反转（图 10）。

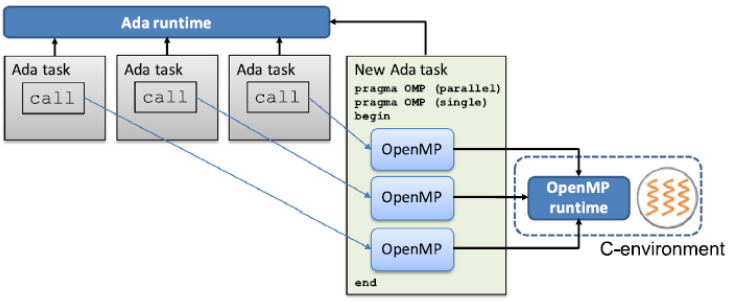


图 10。Ada 与 OpenMP 的模板化执行集成 [64]。

由于 Ada 中的并行构造与现有并发模型相集成，Ada 任务的实时

-时间调度（优先级、截止时间、执行时间等）仍然适用，完整的 Ada 调度模型也同样适用。仍需进行一些澄清并补充文档，例如，当一个高优先级

任务抢占低优先级任务的并行执行时，如果并行计算在所有核心上都被抢占，即使高优先级任务并未使用这些核心。

此外，如代码清单 23 所示，也可以定义具有并行性的周期性周期实时任务（非周期/偶发任务也类似）。因此，该模型使程序员能够轻松地使用分叉/汇合模型构建并行实时应用，并借助同时支持并发和并行性的 Ada 运行时，或使用实现代码清单 18 中所讨论方法的 OpenMP。

6.2 通信与同步

由于并行执行已集成到并发模型中，因此可以使用受保护对象（或原子操作，如代码清单 23 所示）来实现并行执行之间的数据共享。清单 24 给出了一个示例：一个任务在数组中进行并行搜索，并可能更新受保护对象中的计数器。该示例还展示了如何使用优先级上限协议，以确保不会发生优先级反转，并在其他任务使用该对象时保证执行不会死锁（优先级上限_优先级 ≥某些_优先级）。

```
受保护 Counter
        with 优先级 => 上限_优先级;
is
        过程 递增;
        函数 获取 返回 自然数;
私有
        计数: 自然数 := 0;
结束 Counter;
受保护 主体 Counter 为
        过程 递增 是
        开始
                计数 := @ + 1;
        结束 递增;
        函数 获取 返回 自然数 是
        开始
                返回 计数;
        结束 获取;
结束 Counter;

任务 执行 具有 优先级 => Some_优先级;
任务体 执行 是
-- 声明
开始
-- 初始化
        -- ...

        parallel
对 Trace 循环中的元素
                如果 元素 = 0 则
                Counter.递增;
                结束 如果;
结束 循环;
        -- ...
结束 执行;
```

清单 24。并行构造中的数据共享。

请注意，反过来的情况，即在受保护对象内部派生并行执行，是不允许的。如果执行受保护子程序时遇到并行构造，那么并行构造中原本会成为独立逻辑控制线程的每个元素，都会以任意顺序在执行受保护操作的逻辑控制线程上执行 [43] （第 9.5.1] 节）。这是因为

此限制与持有对象锁期间在不同线程中处理计算可能带来的复杂性有关。

并行逻辑控制线程之间无法进行同步，因为不允许执行可能阻塞的操作 [43]（第 5.1 节）。这是更安全的模型设计所带来的结果（除非提供阻塞注解，否则编译时可能无法检测到），因为设计意图是由分叉/汇合执行来提供所有同步。

6.3 映射与调度控制

Ada标准规定，每个并行段（数据块）都配有自己的逻辑控制线程，但未规定这些逻辑控制线程如何映射到底层操作系统，只规定它们具有与 Ada 任务相同的调度属性。在 [65] 中，基于抽象执行器的概念，为底层执行行为建立了一个模型；抽象执行器负责在平台上实际执行 Ada 任务。该抽象旨在规定实现（编译器和运行时）必须为并行执行提供哪些进展保证，同时不限制实现方式：

- 即时进展——当核心可用时，已准备好执行的逻辑控制线程可以并行执行直至完成（仅受空闲核心数量的限制）。
- 最终进展——当核心可用时，已准备好的逻辑控制线程可能需要等待执行器可用，但可以保证最终会有执行器可用，从而使逻辑控制线程最终得以执行。
- 有限进展——即使核心可用，已准备好的逻辑控制线程也可能需要等待执行器可用，而且运行时不保证最终会有执行器可用。这意味着执行器数量有限；当逻辑控制线程阻塞时，执行器可能也会阻塞 [15]。

随后证明，该模型适用于实时系统：逻辑控制线程会在启动执行的同一执行器中运行直至完成，不过该执行器可能会被优先级更高（或截止时间更近）的执行器抢占；如果任务的预算/时间片耗尽，也可能被优先级/截止时间相同的执行器抢占。

结合 Ada实时系统附录中定义的调度机制，可以采用全局工作守恒调度方法 [15], ，并可使用现有的实时分析方法对其进行分析。

不过，所提出的某些机制并未针对并行模型进行考虑，目前仍仅适用于不具备细粒度并行性的 Ada 任务 [66]。具体而言，仍有一些问题尚待解决：

- 并行执行情况下，执行时间度量与控制的语义。
- 在并行构造中更改实时属性的语义和影响尚未定义，并行更改可能导致不良影响。

[15] 由于决定不允许在并行计算中发生阻塞，目前尚未使用有限进展。

- Ada任务亲和性和分派域的控制模型适用于任务，因此没有机制可指定一个任务的并行控制线程如何映射到处理器。

- 精细控制分块以及数据块到控制线程的映射，以支持对映射进行静态规范。

最后两点尤为重要，因为目前在程序员层面尚无机制能够控制并行执行的映射或亲和性，也无法通过分区调度定义静态映射方案。只有将映射和分区调度集成到底层运行时中，才能实现这一点，这与 [14] 中针对 OpenMP 采用的方法类似。这仍是一个有待研究的开放课题。

尽管目前已有一些针对 Ada 2022 [63][67], 并行性特性的有限实现，但仍需进一步研究如何将这些特性与该语言现有的实时能力集成；这是应对尚存挑战的关键因素。

7. 结论

本文概述了实时并行编程范式，并阐明了建立坚实基础的必要性。实时系统是一个充满挑战的工作领域，应用范围广泛，并将并发性和时间属性作为系统的基本属性。实时系统编程需要能够描述和管理这些概念的范式，并将其视为一等实体，而不是事后才考虑的问题。实时系统的开发带来了一些特有挑战，其中包括要求编程模型和技术支持对并发活动及时序相关属性的规范，并允许控制应用程序在平台上的执行。随着对细粒度并行执行的支持，这一挑战愈发突出。

这一领域的当前研究源于不同社区团体的活动交汇，例如高性能计算、实时系统和关键系统。论文分析的两种不同方法（OpenMP 和 Ada）是这类跨学科工作的典型例子，其设计决策各不相同，取决于研究工作最初所属的社区。OpenMP 源自高性能计算社区，基于一套完整而复杂的并行编程模型。Ada 则源自关键系统与实时系统社区，为实时系统编程提供了广泛而复杂的支持。

论文还指出了未来研究的潜在主题，尤其是实时属性的规范，以及如何在从编程模型到下层映射与调度层的垂直技术栈中控制并发和并行执行。

致谢

本研究由欧盟 FP7 和 Horizon 2020 研究与创新计划共同资助，资助协议编号分别为 611016（P-SOCRATES 项目）和 871669

（AMPERE 项目）。作者谨此感谢 S. Tucker Taft 和 Tullio Vardanega 对本文早期版本提出的意见。

参考文献

[1] J. A. Stankovic 和 K. Ramamritham。《硬实时系统》。IEEE 计算机学会出版社，1988年。[2] E. A. Lee。《信息物理系统的过去、现在与未来：聚焦模型》。Sensors，2015年，第15卷，第3期，第4837-4869页。[3] G. Buttazzo，《硬实时计算系统：可预测调度算法与应用》，施普林格，2013年，ISBN 978-1461430193[4] R. Wilhelm、J. Engblom、A. Ermedahl、N. Holsti、S. Thesing、D. Whalley、G. Bernat、C. Ferdinand、R. Heckmann、T. Mitra、F. Mueller、I. Puaut、P. Puschner、J. Staschulat 和 P. Stenström，《最坏情况执行时间问题——方法概述与工具综述》，ACM 嵌入式计算系统汇刊（TECS），第7卷，第3期，2008年4月。[5] L.M. Pinho、V. Nélis、P. Meumeu Yomsi、E. Quiñones、M. Bertogna、P. Burgio、A. Marongiu、C. Scordino、P. Gai、M. Ramponi、M. Mardiak，《P-SOCRATES：面向时间关键型众核系统的并行软件框架》，Microprocessors and Microsystems，第39卷，第8期，2015年，https://doi.org/10.1016/j.micpro.2015.06.004。[6] H. Sutter，《免费午餐已结束——软件并发领域的根本性转向》，Dr. Dobb's Journal，30(3)，2005年3月[7] L. Pinho、E. Quiñones、M. Bertogna、L. Benini、J. Carlos、C. Scordino、M. Ramponi，《并行化执行中的时间关键性挑战》，《高性能与实时嵌入式系统》（HiRES 2014）研讨会，与第9届高性能与嵌入式架构及编译器国际会议（HiPEAC 2014）同期举行，2014年1月。[8] N. Wirth，《迈向实时编程规范》。Commun. ACM 20, 8（1977年8月），577–583。https://doi.org/10.1145/359763.359798[9] J. Sifakis，《实时系统建模——挑战与研究方向》。载于：Henzinger, T.A., Kirsch, C.M.（编）《嵌入式软件》。EMSOFT 2001。计算机科学讲义，第2211卷。施普林格，柏林，海德堡。https://doi.org/10.1007/3-540-45449-7_26[10] C.M. Kirsch，《实时编程原理》。载于：Sangiovanni-Vincentelli, A., Sifakis, J.（编）《嵌入式软件》。EMSOFT 2002。计算机科学讲义，第2491卷。施普林格，柏林，海德堡。https://doi.org/10.1007/3-540-45828-X_6[11] K. Lakshmanan、S. Kato 和 R. Rajkumar，《多核处理器上的并行实时任务调度》，2010年 IEEE 第31届实时系统研讨会，2010年，第259-268页，doi: 10.1109/RTSS.2010.42。[12] A. Saifullah、K. Agrawal、C. Lu 和 C. Gill，《广义并行任务模型的多核实时调度》，2011年 IEEE 第32届实时系统研讨会，2011年，第217-226页，doi: 10.1109/RTSS.2011.27。

[13] S. Baruah、V. Bonifaci、A. Marchetti-Spaccamela、L. Stougie 和 A. Wiese，《周期性实时进程的广义并行任务模型》，2012年 IEEE 第33届实时系统研讨会，2012年，第63-72页，doi: 10.1109/RTSS.2012.59。[14] L.M. Pinho、E. Quiñones、M. Bertogna、A. Marongiu、V. Nélis、P. Gai、J. Sancho（编）《高性能与时间可预测嵌入式计算》，River Publishers，2018年7月，https://www.riverpublishers.com/research_details.php?book_id=507，DOI: 10.13052/rp- 9788793609624。[15] S. Baruah、M. Bertogna、G. Buttazzo，《实时系统的多处理器调度》，施普林格，2015年，ISBN 978-3319086972。[16] A. Melani、M. A. Serrano、M. Bertogna、I. Cerutti、E. Quiñones 和 G. Buttazzo，《支持安全关键型 OpenMP 应用的静态调度方法》，2017年第22届亚洲及南太平洋设计自动化会议（ASP-DAC），2017年，第659-665页，doi: 10.1109/ASPDAC.2017.7858399。[17] M. Gharajeh、S. Royuela、L.M. Pinho、T. Carvalho、E. Quiñones，《多核处理器上的启发式任务到线程映射》，第27届 IEEE 新兴技术与工厂自动化国际会议（ETFA 2022），进行中工作专题，2022年7月。[18] C. Maia、P. Yomsi、L. Nogueira、L.M. Pinho，《使用工作窃取的分叉-汇合任务实时半分区调度》，EURASIP 嵌入式系统期刊，文章编号 2017:31，Springer 国际出版公司，2017年，第1-14页。https://doi.org/10.1186/s13639-017-0079-5。[19] R. D. Blumofe、C. E. Leiserson。1999年。通过工作窃取调度多线程计算。《ACM 杂志》46, 5（1999年9月），720-748。https://doi.org/10.1145/324133.324234。[20] V. Nélis、P. Yomsi、L.M. Pinho，《众核架构上并行应用的最坏情况执行时间分析方法》，Euromicro 数字系统设计会议（DSD 2015），2015年8月。葡萄牙丰沙尔，https://doi.org/10.1109/DSD.2015.105。[21] V. Nélis、P. Yomsi、L.M. Pinho，《多核平台上应用程序执行时间的变化性》，第16届最坏情况执行时间分析国际研讨会（WCET 2016），2016年7月，法国图卢兹，https://doi.org/10.4230/OASIcs.WCET.2016.6。[22] S. Baruah，HPC-DAG 任务的可行性分析。《实时系统》58，134-152（2022年）。https://doi.org/10.1007/s11241-022-09380-z。[23] P. Voudouris、P. Stenström 和 R. Pathan，非相关多处理器上并行应用执行时间的界定。《实时系统》58，189-232（2022年）。https://doi.org/10.1007/s11241-021-09375-2。[24] A. Burns、A. Wellings，《实时系统与编程语言：Ada、实时 Java 和 C/实时 POSIX》，第4版，Pearson，2009年，978-0321417459。[25] ISO/IEC 9899:2018，信息技术——编程语言——C。[26] IEEE Std 1003.1-2017，可移植操作系统接口（POSIX(R)）基础规范，第7版，https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/。[27] ISO/IEC 8652:2012，信息技术——编程语言——Ada。

[28] Java 实时规范 2.0，Java 社区进程计划——JSR：Java 规范请求——JSR# 282 详情，https://jcp.org/en/jsr/detail?id=282，最后访问时间：2023年2月。[29] B. Sprunt、L. Sha、J.-P. Lehoczky，1989年。硬实时系统的非周期任务调度。《实时系统期刊》1：27-60。[30] L. Abeni 和 G. Buttazzo，“将多媒体应用集成到硬实时系统中”，发表于第19届 IEEE 实时系统研讨会论文集，马德里，1998年，第4-13页。[31] L. Sha、R. Rajkumar 和 J. P. Lehoczky。1990年。优先级继承协议：一种实时同步方法。IEEE《计算机汇刊》39卷第9期（1990年9月），1175-1185页。https://doi.org/10.1109/12.57058。[32] ISO/IEC JTC1/SC22/WG9，Ada 参考手册，http://www.ada- auth.org/standards/rm12_w_tc1/html/RM-TTL.html，最后访问时间：2023年2月。[33] C. A. R. Hoare，1974年。“监视器：一种操作系统结构化概念”。《ACM 通讯》17卷第10期，DOI：10.1145/355620.361161。[34] J. Pulido、S. Urueña、J. Zamorano、T. Vardanega、J. de la Puente，《使用 Ada 2005 进行分层调度》。载于：Pinho, L.M.、González Harbour, M.（编）《可靠软件技术—Ada-Europe 2006》。Ada-Europe 2006。计算机科学讲义，第4006卷。施普林格，柏林、海德堡。[35] T.P. Baker。实时进程的基于栈的调度。《实时系统》3(1)，1991年3月。[36] A. Burns 和 A. J. Wellings，“可调度性兼容的多处理器资源共享协议—MrsP”，2013年第25届 Euromicro 实时系统会议，2013年，第282-291页，doi：10.1109/ECRTS.2013.37。[37] 最小实时操作系统（Marte OS），https://marte.unican.es/，最后访问时间：2023年2月。[38] X. Qi、D. Zhu、H. Aydin，实时系统的集群调度：利用率界限与运行时开销。《实时系统》47卷，253-284页（2011年）。https://doi.org/10.1007/s11241-011-9121-1。[39] B. Andersson、L.M. Pinho，2010年。使用 Ada 2012 实现基于任务分割的多核实时调度算法。载于：Real, J.、Vardanega, T.（编）《可靠软件技术—Ada-Europe 2010》。Ada-Europe 2010。计算机科学讲义，第6106卷。施普林格，柏林、海德堡。https://doi.org/10.1007/978-3-642-13550-7_4。[40] A. Burns，《Ravenscar 配置文件》。《ACM SIGAda Ada 通讯》XIX (4)：49-52。doi:10.1145/340396.340450。[41] P. Rogers、J. Ruiz、T. Gingold 和 P. Bernardi，《可靠软件技术 Ada-Europe 2017》中的一种新型基于 Ravenscar 的配置文件，Johann Blieberger 和 Marcus Bader（编）（2017），L NCS 10300，Springer-Verlag，第169-183页。[42] OpenMP 应用程序接口，5.2版，2021年11月，https://www.openmp.org/specifications/，最后访问时间：2023年2月。[43] ISO/IEC JTC1/SC22/WG9，Ada 参考手册（Ada 2022年第35版草案），http://www.ada-auth.org/standards/22rm/html/RM-TTL.html，最后访问时间：2023年2月。[44] C. Maia、L. Nogueira 和 L.M. Pinho，2011年。将实时规范 Java 与分叉/汇合相结合：一种基于优先级的模型。发表于第9届实时与嵌入式 Java 技术国际研讨会论文集

系统（JTRES '11）。计算机协会，纽约，美国纽约州，82-86。https://doi.org/10.1145/2043910.2043924。[45] D. Lea。Java 分叉/汇合框架。载于 ACM 2000 年 Java Grande 会议论文集，JAVA '00，第36-43页，纽约，美国纽约州，2000年。ACM。[46] M. Schmid、F. Fritz 和 J. Mottok，“实时系统中的并行编程”，ARCS Workshop 2019；第32届计算系统架构国际会议，2019年。[47] 西门子股份公司，“嵌入式多核构件”，网址：https://embb.io/，最后访问时间：2023年2月。[48] 英特尔，“线程构件”，网址：http://www.threadingbuildingblocks.org，最后访问时间：2023年2月。[49] Stellar Group，“高性能 ParalleX”，网址：https://stellar-group.org/libraries/hpx/，最后访问时间：2023年2月。[50] S. Royuela Alcazar，“OpenMP 高级编译器分析”，博士论文，2018年，http://hdl.handle.net/10803/620642，最后访问时间：2023年2月。[51] J. Fonseca、V. Nélis、G. Raravi、L.M. Pinho，“带条件执行的实时并行应用多DAG模型”，第30届ACM/SIGAPP应用计算研讨会（SAC 2015），2015年4月，嵌入式系统，萨拉曼卡，西班牙，https://doi.org/10.1145/2695664.2695808。[52] M.A. Serrano、S. Royuela、E. Quiñones，《面向关键实时系统的 OpenMP 规范》。载于：de Supinski, B.、Valero-Lara, P.、Martorell, X.、Mateo Bellido, S.、Labarta, J.（编）《面向演进架构的演进式OpenMP》。IWOMP 2018。计算机科学讲义，第11128卷。施普林格，Cham。https://doi.org/10.1007/978-3-319-98521-3_10。[53] M. A. Serrano、A. Melani、R. Vargas、A. Marongiu、M. Bertogna 和 E. Quiñones，“OpenMP4任务模型的时序特征分析”，2015年嵌入式系统编译器、架构与综合国际会议（CASES），2015年，第157-166页，doi：10.1109/CASES.2015.7324556。[54] J. Sun、N. Guan、Y. Wang、Q. He 和 W. Yi，“带绑定任务的 OpenMP 任务系统的实时调度与分析”，载于2017年 IEEE 实时系统研讨会（RTSS），法国巴黎，2017年，第92-103页。[55] UpScale软件开发工具包（UpScale SDK），http://upscale-sdk.github.io，最后访问时间：2023年2月。[56] G. Tagliavini、D. Cesarini 和 A. Marongiu，“借助轻量级 OpenMP 任务机制，在嵌入式众核加速器上释放细粒度并行性”，载于 IEEE并行与分布式系统汇刊，第29卷第9期，第2150-2163页，2018年9月1日，doi：10.1109/TPDS.2018.2814602。[57] Erika Enterprise V2，http://erika.tuxfamily.org/，最后访问时间：2023年2月。[58] G. C. Buttazzo、M. Bertogna 和 G. Yao，“实时系统的有限抢占式调度：综述”，载于 IEEE工业信息学汇刊，第9卷第1期，第3-15页，2013年2月，doi：10.1109/TII.2012.2188805。

[59] S. Michell、B. Moore、L.M. Pinho，《Tasklettes——面向多核处理器上 Ada 的细粒度并行性》，第18届国际可靠软件技术会议（Ada-Europe 2013），2013年6月，第17–34页。德国柏林，https://doi.org/10.1007/978-3-642-38601-5_2。[60] ISO/IEC JTC1/SC22/WG9，《注释版 Ada 参考手册（ Ada 2022 年第 35 版草案）》，http://www.ada-auth.org/standards/22aarm/html/aa-ttl.html，最后访问于2023年2月。[61] Taft、S、Moore、B、Pinho、L、Michell、S，《通过语言扩展在 Ada 中实现安全并行编程》，高完整性语言技术 ACM SIGAda 年度国际会议（HILT 2014），2014年10月。美国波特兰，https://doi.org/10.1145/2663171.2663181。[62] S. Royuela、L.M. Pinho、E. Quiñones，《融合安全性与高性能领域：将 OpenMP 集成到 Ada 中》，欧洲设计、自动化与测试会议（DATE 2018），2018年3月，第1021–1026页。德国德累斯顿。https://doi.org/10.23919/DATE.2018.8342162。[63] S. Tucker Taft，《将 Ada 202X 分层映射到 OpenMP》。Ada Lett. 40, 2（2020年12月），55–58。https://doi.org/10.1145/3463478.3463482。[64] S. Royuela、L.M. Pinho、E. Quiñones，《通过基于模板的执行实现 Ada 与 OpenMP 运行时互操作性》，《系统架构期刊》（JSA），文章编号101702，Elsevier。第105卷，2020年1月14日。https://doi.org/10.1016/j.sysarc.2019.101702。[65] L.M. Pinho、B. Moore、S. Michell、S. T. Taft，《Ada 中细粒度并行性的执行模型》，第20届国际可靠软件技术会议——Ada-Europe 2015，2015年6月。西班牙马德里，https://doi.org/10.1007/978-3-319-19584-1_13。[66] L.M. Pinho、S. Michell，《会议总结：并行与多核系统》，第18届国际实时 Ada 研讨会（IRTAW 2016），发表于 ACM Ada Letters，第36卷，第1期，2016年6月，https://doi.org/10.1145/2971571.2971583。[67] S. T. Taft，“Ada 2022 中的工作窃取调度器”，Ada 中的工作进展报告环节，第26届 Ada-Europe 国际可靠软件技术会议（AEiC 2022），2022年6月14日至17日，比利时根特，https://doi.org/10.1145/3577949.3577964。