文章编号：2097-1974(2026)03-0058-09 DOI：10.7654/j.issn.2097-1974.20260308

基于DAG的自校正最长路径多核调度算法

许兆淳[1]，杨 雨[2]，姜海峰[1]
（1. 北京航天控制仪器研究所，北京，100039；2. 中国航天电子技术研究院，北京，100094）

摘要：平台式惯性导航系统（Platform Inertial Navigation System，PINS）的主控计算机为硬实时多核嵌入式系统，其控制周期直接影响导航精度。针对传统集中式分区调度因装箱问题导致的资源利用率低、计算频率受限等问题，提出一种基于有向无环图（Directed Acyclic Graph，DAG）的自校正最长路径多核调度算法（Self-Correcting Longest-Path DAG Scheduling Algorithm，SLS）。该算法采用有向无环图对具有依赖关系的惯导任务进行建模，构建两阶段闭环结构：静态阶段基于最长路径优先级分配多核并行任务，动态阶段引入自校正机制，利用加权平均法持续修正节点执行时间估计，克服最坏情况执行时间估计误差对调度性能的累积影响。半实物仿真结果表明，相较于近几年惯导领域的其他算法，SLS算法缩短了任务执行时间，提高了多核资源利用率和负载均衡度，有效提升了PINS的系统精度与实时性能。

关键词：惯性导航系统；实时系统；有向无环图；任务调度；同构多核系统
中图分类号：V448.2 **文献标识码**：A

A Self-Correcting Longest-Path DAG Scheduling Algorithm for Multicore Systems

XU Zhaochun[1], YANG Yu[2], JIANG Haifeng[1]
(1. Beijing Institute of Aerospace Control Devices, Beijing, 100039;
2. China Academy of Aerospace Electronics Technology, Beijing, 100094)

Abstract: As the master control computer of the Platform Inertial Navigation System (PINS) constitutes a hard real-time multicore embedded system, its control cycle directly impacts navigation accuracy. To address the issues of low resource utilization and constrained computing frequency resulting from the bin-packing problem inherent in traditional centralized partitioned scheduling, the Self-Correcting Longest-Path DAG Scheduling Algorithm (SLS) is proposed. The algorithm employs directed acyclic graphs (DAGs) to model complex inertial navigation tasks with precedence constraints, constructing a two-stage closed-loop framework of "static planning and dynamic correction". In the static phase, parallel tasks are greedily allocated across multiple cores based on longest-path priorities. The dynamic phase introduces a self-correcting mechanism that utilizes a weighted averaging method to continuously refine node execution time estimates, thereby mitigating the cumulative degradation of scheduling performance caused by worst-case execution time (WCET) estimation errors. Hardware-in-the-loop simulation experiments demonstrate that, compared with recent state-of-the-art algorithms in the inertial navigation field, the SLS algorithm significantly reduces task execution time, enhances multicore resource utilization and load balancing, and effectively improves the system accuracy and real-time performance of PINS.

Keywords: inertial navigation system; real-time system; directed acyclic graph; task scheduling; homogeneous multicore system

0 引 言

惯性导航系统通过感知载体的角运动和线运动，解算载体的姿态、航向信息，实现对载体进行导航或为上层系统提供导航数据。平台式惯性导航系统（Platform Inertial Navigation System，PINS）将运动传感器固定在可自由旋转的平台上，工作时该平台相对惯性坐标系保持静止，一方面这种惯性导航系统精度更高，另一方面其台体具有隔离机壳的角运动的功能。PINS在国家战略科技与军工项目中具有不可替代的关键作用，其主控计算机为硬实时系统，对于实时性具有很高的要求。在PINS中，软件的任务和功能是固定的，周期性地运行在计算硬件上，这个周期

收稿日期：2026-01-07；修回日期：2026-04-16

便是PINS算法的采样周期。对于高精度PINS而言，忽略硬件系统带来的误差，周期的缩短通常能使误差随之减小。然而这种误差抑制的效果存在：数值稳定边界和实时性边界，前者主要取决于PINS主控计算机的机器精度极限和算法固有特性，一般为微秒级，后者则由主控软件的单周期执行时间决定。

本文针对的PINS主控计算机为多核实时系统，随着战略导弹惯性系统向速率平台等新一代架构演进，PINS主控计算机需承担自主导航、自主控制与自主重构等复杂实时任务，其多核计算资源的合理分配与任务可调度性分析成为保障系统性能的关键[1]。实时系统领域关于任务模型和调度算法的研究始于20世纪70年代，为保证具有严格截止期限任务的实时性，Liu等[2]针对单处理器环境提出了速率单调调度（Rate Monotonic，RM）和最早截止时间优先调度（Earliest Deadline First，EDF）两种基于优先级机制的调度算法，并建立了可调度性的充分条件，奠定了单核实时调度的理论基础。近年来，越来越多的学者采用有向无环图（Directed Acyclic Graph，DAG）对多核实时任务建模，该模型下无依赖关系的节点可并行执行，从而区别于传统顺序任务模型[3-5]，具有关键路径、体积、密度与利用率等独特属性[6]，是多核实时系统的理想任务模型。基于DAG实时任务模型，研究人员根据实际应用环境进行了调度理论与最坏情况响应时间（Worst-Case Response Time，WCRT）分析[7-10]、DAG拓扑结构优化[11]与异构/递归等复杂场景算法设计[12-13]等多方面的理论研究，但这些研究对节点实际运行时间的波动关注较少，通常倾向于使用单一的执行时间预测值，以及对调度可实现性的理论分析。在惯性导航领域，国际上的研究主要关注于任务安全性、同构和异构多核系统上的实现以及能效、精度和实时性方面的优化，中国的研究进度正逐渐向国际靠拢。文献［14］基于ZYNQ SoC实现了低成本、实时双天线GNSS/INS组合导航系统，对本文所研究的同构多核INS实时任务调度具有重要的参考价值；文献［15］针对INS中“精度-实时性”权衡的调度问题进行研究，提出了一种根据任务输出误差动态调整执行频率的控制方法，对本文提出的任务执行时间自校正机制具有重要借鉴意义；文献［16］在紧耦合GNSS/INS架构下进行精度提升研究，为本文的精度优化方向提供了启发；文献［17］采用精确时间基准（纳秒级同步）优化调度时序、保障运算周期确定性的技术路径，与本文基于DAG自校正机制优化多核任务执行时间的思路形成对比参照。

本文面向同构四核嵌入式PINS主控计算机，开展实时任务建模与调度研究。该嵌入式系统原有的软件采用集中式分区调度，分为主控、稳定回路和计算3个分区，其中，计算分区使用两个核并行来提高数据吞吐率。这种集中式分区调度方式因装箱问题导致处理器资源利用率低，限制了计算频率与控制精度。为此，本文采用全局调度策略，利用DAG建模子任务依赖关系，提出基于DAG的自校正最长路径多核调度算法（Self-Correcting Longest-Path DAG Scheduling Algorithm，SLS）。相比原有分区调度模式，SLS可有效缩短任务单周期的执行时间，显著提升系统控制精度。

1　任务建模与问题定义

1.1　任务系统DAG建模

SLS算法的运行环境为由m个同构核心组成的多核计算系统。考虑一个包含n个周期性并行任务的任务集$\mathcal{T}=\{\tau_1, \tau_2, \cdots, \tau_n\}$，任务$\tau_i(1 \leqslant i \leqslant n)$是该任务集中的任意任务。考虑这些执行需求的逻辑关系，该任务可表示为一个DAG，$G_i=(V_i, E_i)$，其节点集$V_i=\{v_i^1, v_i^2, \cdots, v_i^p\}$代表不同的执行需求实例，其中$p$是$\tau_i$中节点的总数，而边集$E_i \subseteq V_i \times V_i$代表节点之间的依赖关系。从节点$v_i^j$到节点$v_i^k$的有向边记为$v_i^j \rightarrow v_i^k(1 \leqslant j, k \leqslant p)$，其中$v_i^j$是$v_i^k$的直接前驱，$v_i^k$是$v_i^j$的直接后驱，这意味着$v_i^k$的执行直到$v_i^j$完成后才能开始。定义$\mathrm{pred}(v_i^k)$为$v_i^k$的所有直接前驱集合，$\mathrm{suc}(v_i^j)$为$v_i^j$的所有直接后驱集合。

一个节点可能有0个或多个直接前驱或直接后驱，每个节点v_i^j在时间上的执行需求记作$w(v_i^j)$。对于任意节点$v_i^j \in V_i$，若$\mathrm{pred}(v_i^j)=\varnothing$，则$v_i^j$为起始节点；若$\mathrm{suc}(v_i^j)=\varnothing$，则$v_i^j$为终止节点。记起始节点集为$S_i$，终止节点集为$T_i$，起始节点到终止节点的路径集为$P_i \subseteq S_i \times T_i$。DAG中节点和边的关系如图1所示，图1中$v_i^g$和$v_i^h$是$v_i^j$的直接前驱，$v_i^k$和$v_i^l$是$v_i^j$的直接后驱。

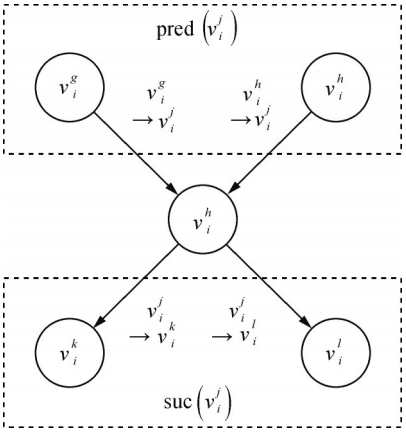


图1 DAG示意

Fig.1 DAG schematic

1.2 问题形式化定义

对于任务集$\mathcal{T}$中给定的DAG任务τ_i和同构m核计算系统，调度问题的目标是生成一个任务调度列表$\mathcal{M}$，将每个节点v_i^j分配到处理器核心$m(v_i^j) \in \{1, 2, \cdots, m\}$，并确定其开始时间$t_s(v_i^j)$和完成时间$t_f(v_i^j)$，使得该分配满足：

a）依赖约束：对于任意边$v_i^j \to v_i^k$，满足$t_s(v_i^k) \geqslant t_f(v_i^j)$，即节点只能在其直接前驱完成后开始；

b）资源约束：任意时刻，同一核心上最多执行一个任务；

c）优化目标：最小化任务完成时间，即$\min \max_{v_i^j \in V_i} t_f(v_i^j)$。

2 调度算法

SLS算法采用“静态规划-动态校正”的两阶段闭环结构，其核心思想是：不仅利用DAG的拓扑结构进行初始优化调度，更通过执行时间的反馈机制持续修正任务权重估计，形成自适应优化闭环。SLS算法包含3个核心模块，如图2所示：

a）最长路径计算模块：基于当前DAG拓扑计算各就绪节点的最长路径优先级；

b）多核分配模块：按优先级贪婪分配至空闲核心；

c）自校正模块：监测实际执行时间，通过加权平均修正节点权重，影响下一轮调度决策。

算法运行时，首先对原始DAG进行拓扑分析，计算各起始节点的最长路径；随后进入调度-执行-校正的循环，当核心空闲时，按最长路径优先级分配就绪任务；任务完成后，利用实际执行时间更新权重估计。在下一周期中，使用修正后的权重重新计算最长路径，实现调度策略的动态优化。

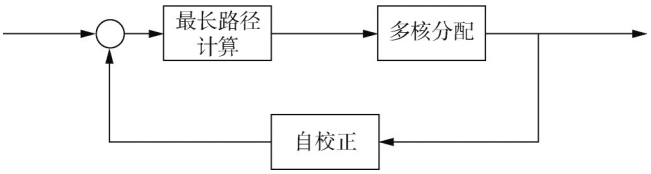


图2 算法核心

Fig.2 Core algorithm

2.1 路径计算与多核分配

在每一轮调度开始前，算法首先识别当前DAG的就绪节点集S，即所有直接前驱已完成的节点。对于每个就绪节点$v_i^j \in S$，计算从该节点到各终止节点的所有路径$p_i^{j,k} \in P_i$的长度$L(p_i^{j,k})$：

$$L(p_i^{j,k}) = \sum_{v_i \in p_i^{j,k}} w(v_i) \tag{1}$$

式中 $p_i^{j,k}$表示从起始节点v_i^j到终止节点v_i^k的路径。对于节点v_i^j，定义其最小最长路径$L_{\min}(v_i^j)$为从该节点出发的所有路径长度的最小值：

$$L_{\min}(v_i^j) = \min_{p_i^{j,k} \in P_s} \{L(p_i^{j,k})\} \tag{2}$$

根据$L_{\min}$值由大到小对就绪节点进行排序，得到调度序列$Q = (v_i^1{}', v_i^2{}', \cdots, v_i^n{}')$满足：

$$L_{\min}(v_i^1{}') \geqslant L_{\min}(v_i^2{}') \geqslant \cdots \geqslant L_{\min}(v_i^n{}') \tag{3}$$

该排序策略确保关键路径上的节点优先执行，从而有效缩短整体完成时间。

当系统中存在空闲核心时，调度器按以下步骤分配任务：

a）从排序序列Q中依次取出节点；

b）将当前节点分配至第一个可用的空闲核心；

c）重复直至所有核心被占用或Q中节点分配完毕。

此贪婪策略确保计算资源优先服务于关键路径任务，同时最大化多核并行度。

2.2 自校正机制

相较于静态DAG调度算法，SLS引入了执行时间反馈闭环，通过持续监测和修正任务执行时间，克服最坏执行时间（Worst-Case Execution Time，WCET）估计误差对调度性能的累积影响。

在任务执行过程中，算法维护一个执行时间记录器。对于每个节点v_i^j，在第n次执行时，记录其实际

执行时间$w_n(v_i^j)$，定义$w(v_i^j)$为节点初始WCET估计值。

当DAG任务所有节点执行完成后，算法根据本轮实际执行数据修正各节点的权重估计。修正公式采用加权平均法：

$$w^*(v_i^j)=\frac{w(v_i^j)+\sum_{l=1}^{n}w_l(v_i^j)}{n+1} \tag{4}$$

式中　$w(v_i^j)$为当前使用的权重值；$\sum_{l=1}^{n}w_l(v_i^j)$为历史n次执行的实际时间总和；$w^*(v_i^j)$为修正后的新权重值。

该策略采用算术平均而非简单替换，可有效抑制单次执行中的随机波动对后续调度的过度影响，同时保证算法能够适应系统运行中的渐进性变化，如系统时间计算模块执行时间随时间推移的增大。

修正后的权重$w^*(v_i^j)$将替代原$w(v_i^j)$，用于下一轮调度的最长路径计算，形成了一个完整的负反馈闭环，如图3所示。

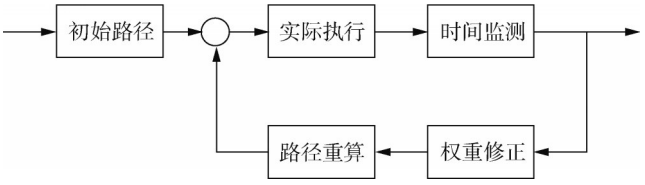


图3　SLS算法负反馈闭环

Fig.3　Negative feedback loop of SLS algorithm

在系统首次运行或任务初次调度时（$n=0$），缺乏历史数据，此时直接使用WCET估计值作为$w(v_i^j)$进行调度。经过若干个周期（通常为3~5个周期）后，修正值$w^*(v_i^j)$将收敛到稳定的实际执行时间均值。

考虑硬实时系统的安全性要求，自校正机制修正后的执行时间不得超过截止期限。为此，SLS采用如下的保守修正策略：

a）若某次实际执行时间$w_l(v_i^j)$显著大于当前$w(v_i^j)$，如超过20%，则触发异常处理，保留原WCET值作为上限，防止因异常值导致后续调度不可行；

b）在负载均衡分析中，使用修正后的均值进行优化，但在可调度性分析中仍保留原始WCET作为安全边界。

2.3　可行性验证

以下为基于DAG的自校正最长路径多核调度算法，给出了SLS的完整伪代码实现。

输入：DAG任务τ=（V，E），核心数m，初始权重W=｛w（v）｝

输出：调度方案M，修正后权重W*

```
初始化：核心状态M=｛空闲｝，执行次数n=0
while V≠∅ do｛
    // 静态规划
    S← 识别就绪节点集（V）
    T← 识别终止节点集（V）
    P← S × T
    LP← calc_Length（P，W）
    for each v_s∈ S do｛
        L_min（v_s）←min ｛LP［p］|p起始于v_s｝
    ｝
    Q← sort（S，L_min，降序）
    for each 空闲核心c ∈ M do｛
        if Q ≠∅ then｛
            v ← pop_front（Q）
            assign（v，c）
            mark_busy（c，v）
        ｝
    ｝
    system_run（）
    wait_until（有节点执行完成）

    // 动态校正准备
    finished ← get_finished_nodes（）
    for each v∈ finished do｛
        record_exec_time（v，actual_time）
        remove_node（V，v）
        ｝
    ｝

    // 自校正阶段（周期结束后）
    for each v_i ∈ original_V do｛
        W* ［v_i］ ←（W ［v_i］ +sum（history_times ［v_i］ ））/（n+1）
    ｝
    return M，W*
```

为估计该算法实际应用的可行性，对SLS算法进行复杂度分析。首先进行时间复杂度分析，算法每轮

迭代至少完成一个节点的调度，故最多执行$|V|$轮，每轮的行为和相应的时间复杂度如表1所示。

表1 SLS算法时间复杂度统计

Tab.1 Time complexity analysis of SLS

行为	时间复杂度
遍历剩余节点识别就绪集	$O(\|V\|)$
基于动态规划计算最长路径	$O(\|V\|+\|E\|)$
排序	$O(\|V\|\log\|V\|)$

单次调度的总时间复杂度为

$$O\left(|V|^2\log|V|+|V|\cdot|E|\right) \tag{5}$$

对于稀疏DAG($|E|\approx|V|$)，复杂度为$O\left(|V|^2\log|V|\right)$；对于稠密DAG$\left(|E|\approx|V|^2\right)$，复杂度为$O\left(|V|^3\right)$。

此后对SLS算法进行空间复杂度分析。算法需存储DAG拓扑结构、节点执行历史（常数周期）及辅助队列，总空间复杂度为$O(|V|+|E|)$。

考虑到PINS主控软件中v通常为数十量级，且调度计算时间在毫秒级，该复杂度满足可调度要求，在PINS主控软件中可行。

3 用例与效果验证

3.1 调度过程示例

给定一个示例任务$\tau_0\in\mathcal{T}_0$，该任务有22个节点，分别表示为DAG节点，如图4所示。其中，每个节点上的标注$x(y)$表示该节点为v_0^x，且$w(v_0^x)=y$个时间单位。

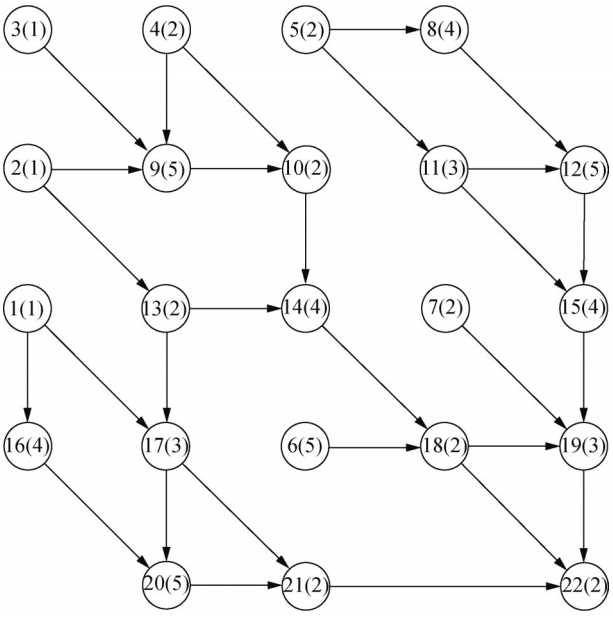


图4 表示为DAG的并行任务τ_0

Fig.4 Parallel task τ_0 represented as DAG

在第一轮调度中，假设对于所有$v_0^x\in\tau_0$，$w(v_0^x)$为其WCET。梳理其中各个起始节点$v_0^s\in S$到终止节点$v_0^t\in T$的路径集P，如图5所示，并计算每条路径$p\in P$的长度$L(p)$，如表2所示。

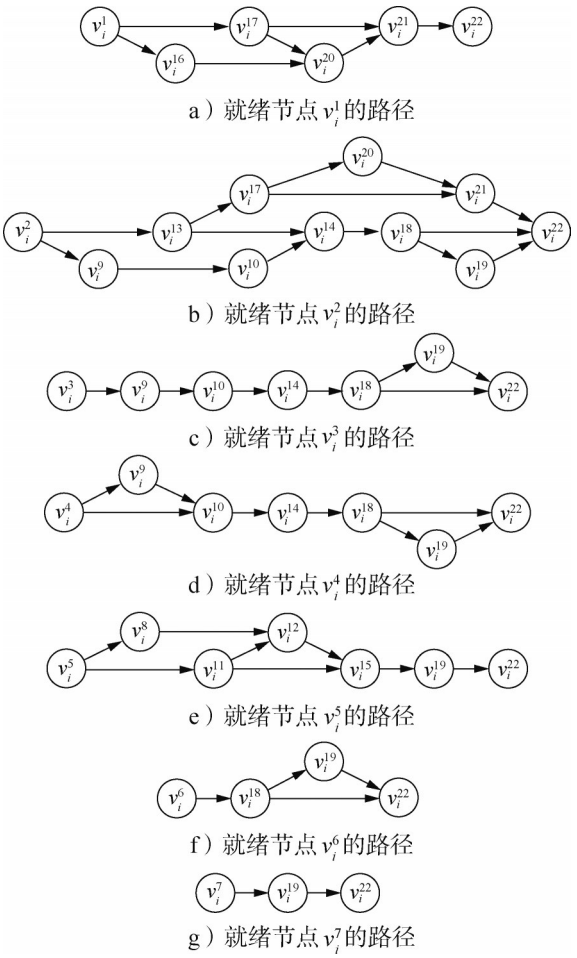


图5 第一次$L(p)$计算路径

Fig.5 Path of the first $L(p)$ calculation

可得DAGτ_0的$L_{\min}(v_0^s)$如下：

$$\begin{aligned}&L_{\min}(v_0^1)=L(p_1^1)=8,\quad L_{\min}(v_0^2)=L(p_2^2)=10,\\&L_{\min}(v_0^3)=L(p_3^1)=16,\quad L_{\min}(v_0^4)=L(p_4^1)=12,\\&L_{\min}(v_0^5)=L(p_5^1)=14,\quad L_{\min}(v_0^6)=L(p_6^1)=9,\\&L_{\min}(v_0^7)=L(p_7^1)=7\end{aligned} \tag{6}$$

对$L_{\min}(v_0^s)$排序，可得：

$$\begin{aligned}&L_{\min}(v_0^3)>L_{\min}(v_0^5)>L_{\min}(v_0^4)>L_{\min}(v_0^2)>\\&L_{\min}(v_0^6)>L_{\min}(v_0^1)>L_{\min}(v_0^7)\end{aligned} \tag{7}$$

则有：

$$Q_0=(v_0^3,v_0^5,v_0^4,v_0^2,v_0^6,v_0^1,v_0^7) \tag{8}$$

之后，遍历序列Q_0，将v_0^3，v_0^5，v_0^4，v_0^2依次分配到1号至4号核心上。至此，便完成了第一轮节点调度，此后算法将等待多核系统再次出现空闲处理器核心。

表2　第一次$L(p)$计算

Tab.2　The first $L(p)$ calculation

就绪节点	$p \in P$	$L(p)$
v_i^1	$p_1^1 = v_i^1 \to v_i^{17} \to v_i^{21} \to v_i^{22}$ $p_1^2 = v_i^1 \to v_i^{17} \to v_i^{20} \to v_i^{21} \to v_i^{22}$ $p_1^3 = v_i^1 \to v_i^{16} \to v_i^{20} \to v_i^{21} \to v_i^{22}$	$L(p_1^1) = 8$ $L(p_1^2) = 13$ $L(p_1^3) = 14$
v_i^2	$p_2^1 = v_i^2 \to v_i^{13} \to v_i^{17} \to v_i^{20} \to v_i^{21} \to v_i^{22}$ $p_2^2 = v_i^2 \to v_i^{13} \to v_i^{17} \to v_i^{21} \to v_i^{22}$ $p_2^3 = v_i^2 \to v_i^{13} \to v_i^{14} \to v_i^{18} \to v_i^{22}$ $p_2^4 = v_i^2 \to v_i^9 \to v_i^{10} \to v_i^{14} \to v_i^{18} \to v_i^{22}$ $p_2^5 = v_i^2 \to v_i^{13} \to v_i^{14} \to v_i^{18} \to v_i^{19} \to v_i^{22}$ $p_2^6 = v_i^2 \to v_i^9 \to v_i^{10} \to v_i^{14} \to v_i^{18} \to v_i^{19} \to v_i^{22}$	$L(p_2^1) = 15$ $L(p_2^2) = 10$ $L(p_2^3) = 11$ $L(p_2^4) = 16$ $L(p_2^5) = 14$ $L(p_2^6) = 19$
v_i^3	$p_3^1 = v_i^3 \to v_i^9 \to v_i^{10} \to v_i^{14} \to v_i^{18} \to v_i^{22}$ $p_3^2 = v_i^3 \to v_i^9 \to v_i^{10} \to v_i^{14} \to v_i^{18} \to v_i^{19} \to v_i^{22}$	$L(p_3^1) = 16$ $L(p_3^2) = 19$
v_i^4	$p_4^1 = v_i^4 \to v_i^{10} \to v_i^{14} \to v_i^{18} \to v_i^{22}$ $p_4^2 = v_i^4 \to v_i^9 \to v_i^{10} \to v_i^{14} \to v_i^{18} \to v_i^{22}$ $p_4^3 = v_i^4 \to v_i^{10} \to v_i^{14} \to v_i^{18} \to v_i^{19} \to v_i^{22}$ $p_4^4 = v_i^4 \to v_i^9 \to v_i^{10} \to v_i^{14} \to v_i^{18} \to v_i^{19} \to v_i^{22}$	$L(p_4^1) = 12$ $L(p_4^2) = 17$ $L(p_4^3) = 15$ $L(p_4^4) = 20$
v_i^5	$p_5^1 = v_i^5 \to v_i^{11} \to v_i^{15} \to v_i^{19} \to v_i^{22}$ $p_5^2 = v_i^5 \to v_i^8 \to v_i^{12} \to v_i^{15} \to v_i^{19} \to v_i^{22}$ $p_5^3 = v_i^5 \to v_i^{11} \to v_i^{12} \to v_i^{15} \to v_i^{19} \to v_i^{22}$	$L(p_5^1) = 14$ $L(p_5^2) = 20$ $L(p_5^3) = 19$
v_i^6	$p_6^1 = v_i^6 \to v_i^{18} \to v_i^{22}$ $p_6^2 = v_i^6 \to v_i^{18} \to v_i^{19} \to v_i^{22}$	$L(p_6^1) = 9$ $L(p_6^2) = 12$
v_i^7	$p_7^1 = v_i^7 \to v_i^{19} \to v_i^{22}$	$L(p_7^1) = 7$

在$t = 1$时，v_0^2和v_0^3执行完毕。删去这两个节点之后，得到DAG τ_0'如图6所示，该DAG相比于τ_0，增加了就绪节点v_0^{13}，减少了就绪节点v_0^3，v_0^5，v_0^4，v_0^2，并删除了v_0^2，v_0^3。

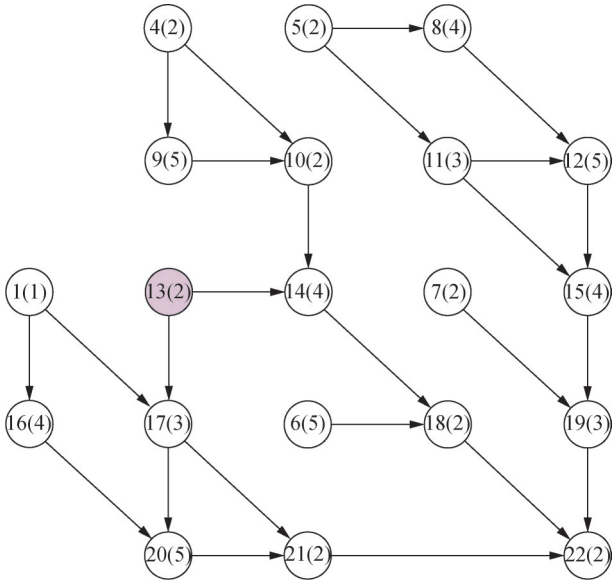


图6　DAG τ_0'

Fig.6　DAG τ_0'

为执行新增就绪节点v_0^{13}的调度，需要对该子任务计算$L(p)$，如图7和表3所示。

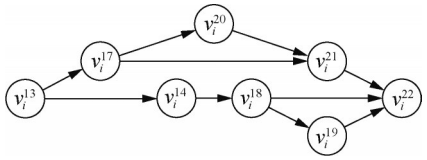


图7　第二次$L(p)$计算路径

Fig.7　Path of the second $L(p)$ calculation

表3　第二次$L(p)$计算

Tab.3　Second $L(p)$ calculation

就绪节点	$p \in$ P	$L(p)$
v_i^{13}	$p_{13}^1 = v_i^{13} \to v_i^{17} \to v_i^{20} \to v_i^{21} \to v_i^{22}$ $p_{13}^2 = v_i^{13} \to v_i^{17} \to v_i^{21} \to v_i^{22}$ $p_{13}^3 = v_i^{13} \to v_i^{14} \to v_i^{18} \to v_i^{22}$ $p_{13}^4 = v_i^{13} \to v_i^{14} \to v_i^{18} \to v_i^{19} \to v_i^{22}$	$L(p_{13}^1) = 14$ $L(p_{13}^2) = 9$ $L(p_{13}^3) = 10$ $L(p_{13}^4) = 13$

可得$L_{\min}(v_0^{13}) = 9$，从而对DAGτ_0'的就绪节点有$L_{\min}(v_0^6) = L_{\min}(v_0^{13}) > L_{\min}(v_0^1) > L_{\min}(v_0^7)$，于是：

$$Q_0' = (v_0^6, v_0^{13}, v_0^1, v_0^7) \tag{9}$$

由于此时空闲出的处理器核心为1号和4号，因此把v_0^6放在1号核心上，v_0^{13}放在4号核心上，从而完成了第二轮节点调度。

四核计算系统第一次运行示例DAG任务τ_0的时序如图8所示，横轴表示从任务释放起经历的时间单位数。

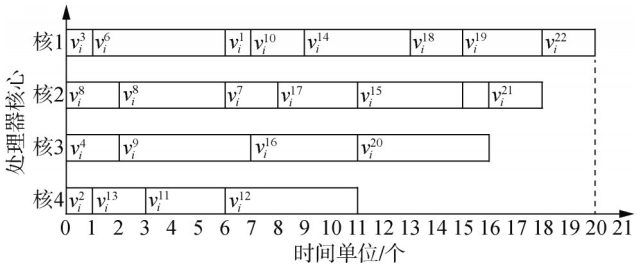


图8　DAG τ_0的运行时序

Fig.8　Running sequence of DAG τ_0

假设在第一个周期中，节点v_0^{13}的实际执行时间为3而非估计的2，则根据式（4），下一轮调度时有：

$$w^*(v_0^{13}) = (2 + 3)/2 = 2.5 \tag{10}$$

这将影响后续包含v_0^{13}的路径长度计算，使调度器更准确地评估通过v_0^{13}的路径实际长度。具体自校正效果见3.2.2节。

3.2　调度效果验证

非抢占式最短处理时间优先（Shortest Processing

Time，SPT）调度算法是一种在单核和多核实时系统中广泛应用的经典调度算法。为验证本算法的先进性，采用SPT算法对DAG任务τ_i进行调度，并将调度结果与上文得到的调度结果进行对比。

非抢占式SPT算法以集合$\tau=\{v_1, v_2, \cdots, v_n\}$来描述任务，其中任务的每个节点被描述为二元组$v_i=(r_i, p_i)$，$r_i\in\mathbb{R}>0$为$\tau_i$到达时间，$p_i\in\mathbb{R}>0$为$v_i$处理时间，$p_i$在调度前已知。定义$t$时刻的就绪节点集为

$$\mathcal{Q}(t)=\{v_i\in\tau | r_i\leqslant t且v_i尚未完成\} \tag{11}$$

该算法定义了一个优先级函数π：$\tau\to\mathbb{R}$如下：

$$\pi(v_i)=p_i \tag{12}$$

该函数赋予v_i以优先级p_i，v_i的处理时间越短，其优先级p_i越高。在任意时刻t，若处理器空闲，则调度器以如下规则选择节点v^*：

$$v^*=\arg\min_{v_i\in\mathcal{Q}(t)}\pi(v_i)=\arg\min_{v_i\in\mathcal{Q}(t)}p_i \tag{13}$$

其中，$\arg\min_{v_i\in\mathcal{Q}(t)}$表示在就绪节点集中选择使目标函数取最小值的节点。

3.2.1 第一轮调度

对于图4所示DAG任务τ_0，考虑节点之间的依赖关系，采用SPT算法进行调度，得到的运行时序如图9所示。

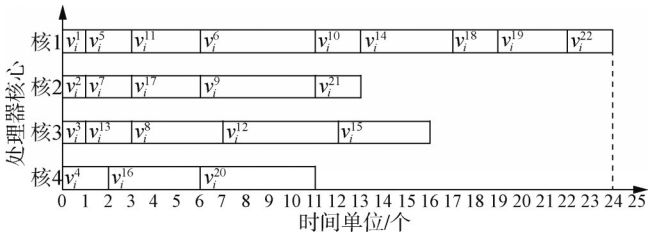


图9 任务τ_0在SPT算法下的运行时序

Fig.9 Running sequence of task τ_0 under SPT algorithm

可见任务完成运行需要$t_{\mathrm{SPT}}=24$个时间单位，而由图8可知，在本文提出的调度算法下，任务完成运行只需$t_{\mathrm{proposed}}=20$个时间单位，运行时间显著缩减，缩减程度为

$$\frac{t_{\mathrm{SPT}}-t_{\mathrm{proposed}}}{t_{\mathrm{SPT}}}\times 100\%\approx 16.7\% \tag{14}$$

任务运行期间核m的空闲时间为ρ^m，由任务运行时序图可以得出，任务τ_i在SPT调度算法下空闲率β_{SPT}与本文提出算法下的空闲率$\beta_{\mathrm{proposed}}$分别为

$$\begin{aligned}\beta_{\mathrm{SPT}}&=\frac{\sum\rho^m_{\mathrm{SPT}}}{4\times t_{\mathrm{SPT}}}=32.3\%\\ \beta_{\mathrm{proposed}}&=\frac{\sum\rho^m_{\mathrm{proposed}}}{4\times t_{\mathrm{proposed}}}=20\%\end{aligned} \tag{15}$$

可见，SLS算法对于多核系统资源的利用效率显著高于SPT算法。

3.2.2 后续轮调度

从第二轮调度开始，DAG节点的$w(v_0^x)$会根据实际运行时间进行修正，调度结果也会随之变化，从而逐渐缩短运行时间，最终逼近一个稳定值$w^*(v_0^x)$。10轮调度之后，SPT算法、传统DAG最长路径调度算法和本文提出的SLS算法下，τ_0的运行时间如图10所示，可以看出本文提出的SLS算法在自校正机制方面的优势。

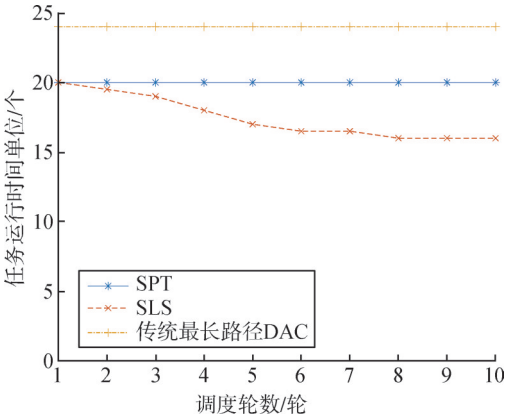


图10 DAG τ_0运行时间

Fig.10 Running time of DAG τ_0

4 仿真试验

将本文调度算法应用于PINS主控软件，进行多核任务队列优化，将优化后的软件在PINS主控计算机上运行，并测试其运行完成时间，从而验证上述方法的有效性。本试验中，运行完成时间指的是一个周期内软件从开始运行时刻到完成运行时刻之间的时长。

试验采用半实物仿真的方式进行，通过嵌入式开发板模拟PINS主控计算机。测试硬件环境由嵌入式开发板、测试计算机、仿真器和示波器构成。嵌入式开发板核心为一片四核6713 DSP，测试计算机处理器为Intel Core i5-8265U CPU，二者经由仿真器连接，采用JTAG+USB接口，示波器直接连接DSP的GPIO引脚。测试计算机运行Windows 7系统，安装CCS 3.3 IDE进行调试，该IDE的性能分析器可以在嵌入式软件运行过程中分析任务的执行时序。示波器可监测GPIO的高低电平，通过与特定软件代码结合，可以分别表征每个DSP核心中软件的运行状态。测试系统

硬件环境的构成如图11所示。

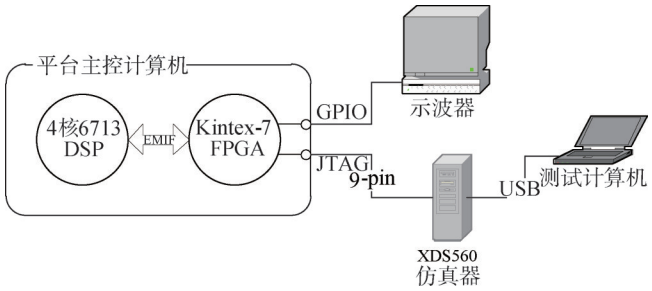


图11　测试硬件系统

Fig.11　Hardware system under test

试验中使用示波器监测的软件运行状态为任务的运行完成时间，通过在任务首尾添加代码脚本将其映射为GPIO引脚电平，规定当程序尚未运行结束时，电平为高，当程序运行结束时，电平为低。试验步骤如下：

a）首先在各个核心主控软件任务的起始和终止位置加入GPIO高低电平切换语句，为确保正确表征不同核心的并行任务运行时序，将四个核心的任务分别绑定到4个不同的GPIO上。修改完成后进行编译。

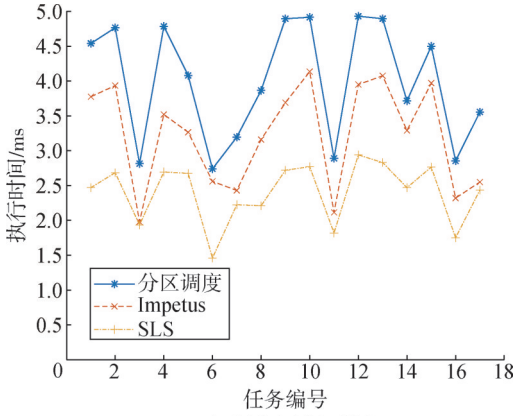
b）按照图11方式连接测试硬件系统，并在CCS 3.3 IDE中完成测试计算机与开发板的连接。

c）使用CCS 3.3将平台主控软件按照设计的顺序分别加载到DSP四个核心的RAM中，观察示波器波形，并测量、记录四个核心中任务的运行时序。

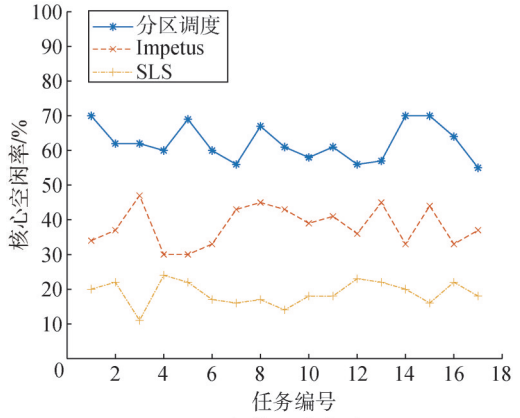
本试验采用的任务集为本文所针对的PINS主控计算机上应用的17个典型主控任务，这些任务可概括为上位机指令处理任务、系统电源管理任务、系统状态监测任务、温度控制任务、框架跟踪任务、标定任务、对准任务、导航任务等。试验环境下软件的运行周期为5 ms。

为了验证本算法在惯性导航领域的优越性，本试验将PINS主控软件分别应用本文所提出的SLS调度算法和两种近年提出的调度策略，并对运行效果进行对比。文献［16］采用了松耦合/紧耦合集成逻辑下的分区调度策略，代表当前惯性导航领域的主流做法；文献［15］提供了另一种应对执行时间波动的方案，即启发式的Impetus算法，作为本文提出的SLS算法中的自校正机制的对照。这两种算法均为在导航领域经过验证的调度算法，面临着与PINS相似的离散周期性计算工况，均可满足PINS任务调度的要求。

首先应用分区调度策略，对运行完成时间及各个核的负载密度进行测量，然后分别将调度算法改为Impetus和SLS，再对改进后的任务时序指标进行测量，得出17个PINS主控任务在不同调度算法的调度下，运行时间和核心空闲率表现见图12。



a）任务运行时间



b）核心空闲率

图12　试验结果数据

Fig.12　Plot of experimental results

由图12可见，在分区调度中，由于装箱问题的固有限制，其任务执行的最长时间将近占满5 ms，其核心空闲率也较高。Impetus算法采用启发式调度策略，且是全局调度的，从而运行时间相较于分区调度明显缩短。而本文提出的SLS调度算法可以使单个任务在多核系统中的最长执行时间被进一步压缩至3 ms左右，且各核负载更为均衡。在试验所示的周期长度范围内，SLS算法相较于分区调度算法的执行时间压缩效果仍可实现显著的精度提升。试验表明，在本文的高精度PINS中，这种周期缩短可带来约8%的精度提升。

5　结　论

综上所述，本文设计的SLS调度算法通过构建系统实时调度模型，并将DAG模型中的约束条件转化

为多核系统的时间约束并行执行，充分利用有限多核系统的特点，有效缩短了任务的执行时间，提高了多核系统负载的均衡度，对PINS的精度提升提供了助力。本算法在PINS系统中得到了验证，经论证该算法对实时性边界的突破能够对PINS运算精度起到提升作用，但是由于系统架构和所运行的算法不同，这种精度提升作用是否可扩展至捷联式惯导系统和组合导航系统有待进一步验证。

参 考 文 献

[1] 焦泽德, 魏宗康, 高荣荣. 面向战略导弹的新一代惯性系统[J]. 导航与控制, 2021, 20(4): 1-8.
JIAO Zede, WEI Zongkang, GAO Rongrong. A new generation inertial system for strategic missile[J]. Navigation and Control, 2021, 20(4): 1-8.

[2] LIU C L, LAYLAND J W. Scheduling algorithms for multiprogramming in a hard-real-time environment[J]. Journal of the ACM, 1973, 20(1): 46-61.

[3] BARUAH S, BONIFACI V, MARCHETTI-SPACCAMELA A, et al. A generalized parallel task model for recurrent real-time processes[C]. Piscataway: 2012 IEEE 33rd Real-Time Systems Symposium (RTSS 2012), 2012.

[4] VERUCCHI M, OLMEDO I S, BERTOGNA M. A survey on real-time DAG scheduling, revisiting the global-partitioned infinity war[J]. Real-time Systems, 2023, 59(3): 479-530.

[5] ZHAO S, DAI X, BATE I, et al. DAG scheduling and analysis on multi-core systems by modelling parallelism and dependency[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(12): 4019-4038.

[6] GUAN F, QIAO J, HAN Y. DAG-Fluid: A real-time scheduling algorithm for DAGs[J]. IEEE Transactions on Computers, 2021, 70(3): 471-482.

[7] 韩丁. 两阶段可调度性分析优化方法研究[D]. 沈阳: 东北大学, 2023.
HAN Ding. Research on two-stage schedulability analysis optimization method[D]. Shenyang: Northeastern University, 2023.

[8] 常爽爽, 赵栩锋, 刘震宇, 等. 基于异构多核的多类型DAG任务的响应时间分析[J]. 计算机学报, 2020, 43(6): 1052-1068.
CHANG Shuangshuang, ZHAO Xufeng, LIU Zhenyu, et al. Response time analysis for multi-type DAG tasks based on heterogeneous multi-core[J]. Chinese Journal of Computers, 2020, 43(6): 1052-1068.

[9] 韩美灵, 邓庆绪, 张天宇, 等. 基于G-EDF的DAG并行任务多核响应时间分析[J]. 东北大学学报(自然科学版), 2019, 40(3): 315-320.
HAN Meiling, DENG Qingxu, ZHANG Tianyu, et al. Response time analysis of DAG parallel tasks on multi-core based on G-EDF[J]. Journal of Northeastern University (Natural Science), 2019, 40(3): 315-320.

[10] 李峰, 毕冉, 马野, 等. 带优先级DAG实时任务图模型的响应时间分析[J]. 计算机学报, 2024, 47(12): 2909-2924.
LI Feng, BI Ran, MA Ye, et al. Response time analysis of DAG real-time task graph model with priority[J]. Chinese Journal of Computers, 2024, 47(12): 2909-2924.

[11] 吴毓龙. 多核实时系统DAG任务划分调度算法研究[D]. 哈尔滨: 哈尔滨工业大学, 2024.
WU Yulong. Research on DAG task partitioning and scheduling algorithms for multi-core real-time systems[D]. Harbin: Harbin Institute of Technology, 2024.

[12] 骆亮. 多核平台两级抢占式固定优先级DAG递归调度[D]. 哈尔滨: 哈尔滨工业大学, 2023.
LUO Liang. Two-level preemptive fixed-priority recursive scheduling of DAG on multi-core platform[D]. Harbin: Harbin Institute of Technology, 2023.

[13] 左俊杰, 肖锋, 黄姝娟, 等. 一种新的异构多核平台下多类型DAG调度方法[J]. 计算机应用研究, 2025, 42(2): 514-518.
ZUO Junjie, XIAO Feng, HUANG Shujuan, et al. A new scheduling method for multi-type DAG on heterogeneous multi-core platform[J]. Application Research of Computers, 2025, 42(2): 514-518.

[14] CUI H, WANG Q, WANG S, et al. A ZYNQ-based low-cost real-time dual-antenna GNSS/INS integrated navigation system for excavator[J]. Measurement Science and Technology, 2025, 36(1): 015014.

[15] NASRI M, KARGAHI M, MOHAQEQI M. Scheduling of accuracy-constrained real-time systems in dynamic environments[J]. IEEE Embedded Systems Letters, 2012, 4(2): 45-48.

[16] ZHAO L, BLUNT P, YANG L, et al. Performance analysis of real-time GPS/Galileo precise point positioning integrated with inertial navigation system[J]. Sensors, 2023, 23(5): 2396.

[17] 郝现伟, 王报华, 黄俊木, 等. 运载火箭电气系统高精度时间同步FC-AE-1553总线数据调度算法研究[J]. 导弹与航天运载技术(中英文), 2026(1): 92-98.
HAO Xianwei, WANG Baohua, HUANG Junmu, et al. Research on high precision time synchronous FC-AE-1553 bus data scheduling algorithm for launch vehicle electrical system[J]. Missiles and Space Vehicles, 2026(1): 92-98.

作 者 简 介

许兆淳（2000—），男，硕士研究生，主要研究方向为惯性导航系统及其应用。

杨 雨（1967—），男，研究员，主要研究方向为惯性系统总体设计。

姜海峰（1987—），男，研究员，主要研究方向为惯性导航系统软件总体设计。