# 论文信息

- 标题：Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing
- 时间：2002.03
- 期刊：IEEE TPDS
- 算法名称：HEFT & CPOP
- 意义：在异构平台上提出了相对较新的调度算法，能够提供较好的性能表现和较快的调度时间。
- 作者：Topcuoglu, H; Hariri, S; Wu, Min-You

DAG的本质其实就是去解决2个子问题：

1. 如何定义不同任务的优先级；

2. 如何选择服务器。

# 一、问题场景

异构计算场景：

计算环境由 $q$ 个异构处理器构成集合 $Q$ ，这些节点是全连接的。
所有处理器之间的通信是非竞态的。
计算和传输可以同时进行。
对于给定应用程序的子任务执行是非抢占式的，即除非该子任务主动放弃处理器，其他进程不得剥夺其对处理器的占用。
主动放弃处理的的情况包括：

- 子任务执行完毕，正常退出；
- 子任务需要进行通信操作；
- 子任务运行出错；
- 处理器遇到不可挽回的故障。

异构的处理器对相同的任务可能存在不同的执行时间。
异构处理器发起一次通信需要消耗时间；数据在处理器之间传播有速率限制。
当子任务通信发生在位于同一个处理器上时，通信时间的开销可以忽略不计。

本文所研究的对象是具有依赖性的程序。这些程序需要在异构的平台上执行，每个异构执行单元执行相同的任务时需要消耗不同的时间；处理单元之间的通信需要一定的开销。

# 二、研究对象

HEFT:

考虑了两个被调度到同一个处理器上的任务之间的“空隙”，这些“空隙”可以被其他与后续任务无关的其他并行任务利用起来。

CPOP:

考虑了影响任务执行总体时间的其实是最长的关键路径，通过最小化这个关键路径可以获取相对较短的执行时间。同时，将这些处于关键路径上的任务调度到同一个处理器上可以减少任务之间的通信开销。

创新点在于：

- 与某些工作相比，本文提供的方法是专注于异构平台的调度算法；
- 与其他工作相比，本文提供的方法能够提供更低的时间复杂度；
- 与其他工作相比，本文提供了更全面的实验评估环节：使用参数化生成的依赖任务和现有的真实依赖任务进行测试。

# 三、数学模型

(经典的模型)

依赖型任务可以被描述为一个 DAG 图 $G = (V, E)$ ，其中：

- $V$: 节点（子任务）集合，大小为 $v$
- $E$: 边（依赖关系）集合，其中的元素可以表示为 $(i, j)$ ，意思是任务 $n_j$ 必须在 $n_i$ 完成之后才能执行

data 矩阵描述了两个子任务之间需要传递的数据量，大小为 $v \times v$，其中元素可表示为 $\text{data}_{i,k}$ 表示任务 $n_i$ 需要向任务 $n_k$ 传输的数据量

如果一个任务没有前置任务，该节点就称为入点；如果一个任务没有后继任务，该节点就称为出点。
对于给定的 DAG 图，可以不失一般性地假设其只有一个入点和出点。
（如果 DAG 图有多个入点，可以构造一个无计算要求的伪入点作为所有入点的前置任务，对于多个出点也可以这么做）

$W$ 是一个大小为 $v \times q$ 的矩阵，其中的元素 $w_{i,j}$ 代表了子任务 $n_i$ 在处理器 $q_j$ 上的预计执行时间。

定义平均子任务执行时间为

$$\overline{w_i} = \sum_{j=1}^{q} w_{i,j}/q$$

$B$ 是一个大小为 $q \times q$ 的矩阵，其中的元素 $B_{i,j}$ 代表了两个处理器 $q_i$ 和 $q_j$ 之间的数据传输速率；
$L$ 是一个长度为 $q$ 的向量，其中的一个分量 $L_i$ 代表了处理器 $q_i$ 发起一次通信的开销。

综上可以给出边集中每条边 $(i,j)$ 的通信开销（假设 $n_i$ 被调度在 $p_m$ 上执行， $n_j$ 被调度在 $p_n$ 上执行）

$$c_{i,j} = L_m + \frac{\text{data}_{i,j}}{B_{m,n}}$$

定义平均通信开销为

$$\overline{c_{i,j}} = \overline{L} + \frac{\text{data}_{i,j}}{\overline{B}}$$

$$\overline{L} = \sum_{i=1}^{q} L_i/q$$

$$\overline{B} = \sum_{i=1}^{q}\sum_{j=1}^{q} B_{i,j}/q^2$$

针对每个子任务，定义最早开始时间 EST 和最早完成时间 EFT

$$\text{EST}(n_i, p_j) = \max\{\texttt{avail}[j], \max_{n_m \in \text{Pred}(n_i)}(AFT(n_m) + c_{m,i})\}$$

$$\text{EST}(n_{entry}, p_j) = 0$$

$$\text{EFT}(n_i, p_j) = w_{i,j} + \text{EST}(n_i, p_j)$$

$$\text{Pred}(n_i) = \{n_j | (i,j) \in E\}$$

其中

- $\texttt{avail}[j]$ 指的是处理器 $p_j$ 在完成其他子任务 $n_k$ 之后处于空闲状态的时刻

- $AFT(n_m)$ 指的是任务 $n_m$ 的实际完成时间，与之对应的还有 AST 指的是任务的实际开始时间

程序的完成时间为

$$\text{makespan} = AFT(n_{exit})$$

程序的优化目标为

$$\min \text{makespan}$$

难点在于如何在时间先后顺序的限制下尽可能多地安排工作。

# 四、算法设计

作者首先引入了优先级的概念，该优先级由两部分构成：

$$\text{Priority} = \text{Upward Rank} + \text{Downward Rank}$$

Upward Rank 的定义是：

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in \text{Succ}(n_i)} (\overline{c_{i,j}} + rank_u(n_j))$$
$$rank_u(n_{exit}) = \overline{w_{exit}}$$

$rank_u$ 实际上表征了当前任务到一个出口任务的关键路径长度。如果只考虑计算开销，则其变成静态的 $rank_u^s$ 。

Downward Rank 的定义是：

$$rank_d(n_i) = \max_{n_j \in \text{Pred}(n_i)} (rank_d(n_j) + \overline{w_j} + \overline{c_{j,i}})$$
$$rand_d(n_{entry}) = 0$$

$rank_d$ 实际上表征了从一个入口任务到当前任务开始执行的关键路径（最长路径）长度。

## 1 算法1：HEFT: Hetirogeneous-Earliest-Finish-Time

```
// 初始化
使用平均值初始化计算开销和通信开销;
从出点开始计算所有任务的 rank_u ;
// 使用 rank_u 对所有任务进行优先级排序，排序顺序为非增;
list = sort('rank_u');
while not list.empty():
    n_i = list.pop();
```

```
    // 将 n_i 调度到能够给出最小 EFT 的处理单元上
    EFT_MIN = INFinity;
    EFT_MIN_PROCESSOR = 0;
    for p_k in P:
        // 计算 EFT 的时候要考虑  **两个已经调度的任务之间的可用时隙**
        EFT = EFT(n_i, p_k);
        if EFT < EFT_MIN:
            EFT_MIN = EFT;
            EFT_MIN_PROCESSOR = p_k;
        endif
    endfor
    schedule[n_i] = p_k;
endwhile
```

算法的时间复杂度为 $O(E \times P)$

## 2 算法2：CPOP: Critical-Path-on-a-Processor

```
// 初始化
使用平均值初始化计算开销和通信开销;
从出点开始计算所有任务的 rank_u ;
从入点开始计算所有人物的 rank_d ;
计算所有任务的优先级 priority = rank_u + rank_d ;
|CP| = priority(n_entry);
// 构造关键路径节点集合
S_CP = {n_entry};
n_k = n_entry;
while not isExitTask(n_k) :
    for n_j in Succ[n_k]:
        if priority(n_j) == |CP|:
            S_CP = S_CP + {n_j};
            n_k = n_j;
            breakfor;
        endif
    endfor
endwhile
// 将关键路径上的所有任务调度到能够给出最小 执行时间(w) 的处理单元上
W_MIN = INFinity;
W_MIN_PROCESSOR = 0;
for p_k in P:
    W = 0;
    for n_i in S_CP:
        W = W + w[n_i, p_k];
    endfor
    if W < W_MIN:
```

```
        W_MIN = W;
        W_MIN_PROCESSOR = p_k;
    endif
endfor
// 使用 priority 对所有任务进行优先级排序，排序顺序为非增;
list = sort('priority');
while not list.empty():
    n_i = list.pop();
    if n_i in S_CP:
        schedule[n_i] = W_MIN_PROCESSOR;
    else:
        EFT_MIN = INFinity;
        EFT_MIN_PROCESSOR = 0;
        for p_k in P:
            // 计算 EFT 的时候要考虑  **两个已经调度的任务之间的可用时隙**
            EFT = EFT(n_i, p_k);
            if EFT < EFT_MIN:
                EFT_MIN = EFT;
                EFT_MIN_PROCESSOR = p_k;
            endif
        endfor
        schedule[n_i] = EFT_MIN_PROCESSOR;
    endif
    list.update();
endwhile
```

算法的时间复杂度为 $O(E \times P)$

为了优化总体执行时间，作者从两个角度考虑了优化问题：

- 一个是利用所有可用的调度时间，将任务执行的开始时间从原来的“前序任务执行完毕后”扩展到了“处理单元空闲时”。这里的改动使得一些没有依赖相关性的任务可以更好地进行并行。
- 一个是找到对整体执行时间影响最大地关键路径，将这个关键路径上的任务安排到同一个性能最强的处理单元上。这里的改动避免了关键任务路径上的数据传输开销，可以最大程度上减少其对总体执行时间的影响。

【疑问】 这里对关键路径的优化会不会导致出现新的关键路径?

见 Heuristic Offloading of Concurrent Tasks for Computation-Intensive Applications in Mobile Cloud Computing

# 五、实验结果

实验使用了以下指标对 DAG 的调度结果进行了衡量：

SLR: Schedule Length Ratio

$$SLR = \frac{makespan}{\sum_{n_i \in CP_{MIN}} \min_{p_j \in Q}\{w_{i,j}\}}$$

Speedup: 加速比

$$Speedup = \frac{\min_{P_j \in Q}\{\sum_{n_i \in V} w_{i,j}\}}{makespan}$$

使用以下指标对算法优势进行了衡量：

- 算法获得最好结果的次数；
- 算法的运行时间。

作者在实验过程中使用了随机生成的 DAG 以及现实应用的 DAG 。

随机生成的 DAG 图有以下几个重要参数：

- 子任务数量： $v$ 。
- 形状参数： $\alpha$ 。随机生成的 DAG 图高度符合平均值为 $\lceil \sqrt{v}/\alpha \rceil$ 的均匀分布；每层的节点数量符合平均值为 $\sqrt{v} \times \alpha$ 的均匀分布。（$\alpha \gg 1.0$ 时会生成密集的 DAG ，并行度很高；$\alpha \ll 1.0$ 时会生成稀疏的 DAG ，并行度很低）
- 出度
- 传输计算比：CCR 。 CCR 指的是平均通信时间与平均计算时间的比值。低 CCR 的 DAG 可视作计算密集型的任务。
- 异构参数： $\beta$ 。该参数越高，任务在不同处理器上执行时间差距越大。

$$\overline{w_i} \times \left(1 - \frac{\beta}{2}\right) \leq w_{i,j} \leq \overline{w_i} \times \left(1 + \frac{\beta}{2}\right)$$

实验结果显示，算法给出的调度结果中，排名依次为：

HEFT, CPOP, DLS, MH, LMT

针对 CCR 较大的任务，作者提出了其他可选的策略作为补充，可以提高 HEFT 算法在该场景下的效率。

实验还研究了处理单元数量和效率之间的关系。
总体来说，处理单元数量越多，总理利用率越低。

# 六、启发与思考

优点：

1. 本文的实验设计具有相当的参考价值，在进行 DAG 相关实验设计时可以对照查看；

2. 本文在调研时针对现有方法做了类似发展脉络与研究分支的分类，具有借鉴价值；

3. 本文的从论文标题开始就强调了作者要做的是异构平台上的 `Efficient` 的调度算法，因此作者在设计时就考虑到了算法复杂度的问题，同时在调研过程中也重点考虑了其他调度算法的复杂度问题。